VŠB – Technical University of Ostrava
Faculty of Electrical Engineering and Computer Science
Department of Computer Science

Development of a Hybrid Group Method in Data Handling (GMDH) Network in C++ Framework

Vývoj hybridní Group Method in Data Handling (GMDH) sítě v C++ frameworku

2016 Petr Martínek

VŠB - Technical University of Ostrava
Faculty of Electrical Engineering and Computer Science
Department of Computer Science

Bachelor Thesis Assignment

Student: **Petr Martínek**

Study Programme: B2647 Information and Communication Technology

Study Branch: 2612R025 Computer Science and Technology

Title: Development of a Hybrid Group Method in Data Handling (GMDH) Network in C++ Framework
Vývoj hybridní Group Method in Data Handling (GMDH) sítě v C++ frameworku

The thesis language: English

Description:

This thesis will involve the development and analysis of a Group Method in Data Handling (GMDH) network architecture in C++ language.
Generally, system identification techniques are applied in order to model and predict the behaviors of unknown and very complex systems based on given input-output data. A major difficulty in modeling complex systems is the problem of the researcher introducing prejudices into the model. A.G. Ivakhnenko (1968), introduced a method, based in part on the Rosenblatt Perceptron (Rosenblatt 1958), that allows the researcher to build models of complex systems without making assumptions about the internal workings. GMDH is a self-organizing approach that can overcome some practical limitations in Artificial Neural Networks (ANN). Ivakhnenko's GMDH algorithm constructs a self-organizing model (an extremely high-order polynomial in the input variables) that can be used to solve prediction, identification, control synthesis, and other system problems.

The thesis will include the following:
1. Development of the GMDH architecture in C/C++ framework using the g++ standard.
2. Implementation of Single Value Decomposition (SVD) as regression analysis technique for the calculations of the coefficients of the neurons.
3. Application and verification of the developed GMDH architecture with manufacturing engineering profiling systems.

The final output of the thesis be a functional GMDH Toolkit

References:

[1] Ivakhnenko, A.G. (1968). The Group Method of Data Handling-A rival of the Method of Stochastic Approximation. Soviet Automatic Control, 13 (3), 43-55.
[2] Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain, Psychological Review, 65(3): 386-408.
[3] Onwubolu, G.C. (2008). Design of hybrid differential evolution and group method in data handling networks for modeling and prediction. Information Sciences 178, 3618–3634.
[4] Onwubolu, G. (2009). Hybrid Differential Evolution and GMDH Systems. Springer- Verlag Berlin Heidelberg, SCI 211, 139-191.
[5] Park, H.-S., Park, B.-J., Kim, H, -K., and Oh, S.-K. (2004). Self-organizing polynomial neural networks based on genetically optimized multi-layer perceptron architecture. International Journal of Control,

Automation, and Systems, 2(4), 423- 434.
[6] Nikolaev, N.Y. and Iba, H. (2003). Polynomial Harmonic GMDH Learning Networks for Time Series modelling. Neural Networks, 16, 1527–1540.
[7] Onwubolu, G.C., Sharma, S., Dayal, A., Bhartu, D., Shankar, A., Katafono, K. (2008). Hybrid particle swarm optimization and group method of data handling for inductive modeling. In: Proceedings of International Conference on Inductive Modeling, Kyiv, Ukraine, September 15-19.
[8] Kordik, P. (2006). Fully Automated Knowledge Extraction using Group of Adaptive Models Evolution. PhD Thesis, Dept. of Comp. Sci. and Computers, FEE, CTU Prague, Czech Republic

Extent and terms of a thesis are specified in directions for its elaboration that are opened to the public on the web sites of the faculty.

Supervisor: **doc. MSc. Donald David Davendra, Ph.D.**

Date of issue: 01.09.2015

Date of submission: 29.04.2016

doc. Dr. Ing. Eduard Sojka
Head of Department

prof. RNDr. Václav Snášel, CSc.
Dean of Faculty

I hereby declare that this bachelor's thesis was written by myself. I have quoted all the references I have drawn upon.

Ostrava, April 29, 2016 .....

First and foremost, I would like to express my gratitude to my supervisor doc. Donald Davendra, Ph.D., for giving me the opportunity to work on this very interesting thesis topic. His experience, understanding and patience helped me considerably during my work on this thesis and my bachelor studies as a whole.

I would also like to thank my parents for their loving and caring support and for giving me the opportunity to earn my bachelor degree.

And finally I would like to thank my friends for being there for me.

Abstrakt

Tato bakalářská práce popisuje implementaci Group Method in Data Handling, jejíž síťovou strukturu lze vytvořit pouze se základními znalostmi o řešeném problému. Díky této vlastnosti je vhodná pro optimalizaci pomocí Evolučních Algoritmů. To vytváří dva na sobě skoro nezávislé algoritmy, což dává nové možnosti v oblasti optimalizace a paralelizace. Hlavním cílem této práce je fungující aplikace, jež obsahuje multi platformě optimalizovanou a paralelizovanou implementaci Group Method in Data Handling s Differencialní Evoluci v C++.

Klíčová slova: GMDH, DE, SVD, GMDH paralelizace

Abstract

This thesis describes the implementation of Group Method in Data Handling, where network structure is obtained from a data structure, that can be created with only having the basic knowable of the problem. This makes it suitable to be optimized by Evolutionary Algorithms. This in turn creates two almost independent algorithms, which gives rise to new possibilities, when it comes to optimization and parallelization. A working application that contains cross-platform optimized and parallelized implementation of Group Method in Data Handling with Discrete Differential Evolution in C++ is the main aim of this thesis.

Key Words: GMDH, DE, SVD, GMDH parallelization

Contents

List of symbols and abbreviations

AVG	–	Average
CD	–	Compact Disc
DDE	–	Discrete Differential Evolution
DE	–	Differential Evolution
EA	–	Evolutionary Algorithm
GA	–	Genetic Algorithm
GMDH	–	Group Method in Data Handling
SVD	–	Singular Value Decomposition
VKG	–	Volterra-Kolmogorov-Gabor

List of Figures

List of Tables

Listings

1 Introduction

1.1 Group Method of Data Handling

Group Method of Data Handling (GMDH) was first described by Russian scientist A. G. Ivakhnenko [2]. It is a method which allows to build models of complex systems, without any knowledge about their internal mechanisms. This method creates a model of a system, that was generated based only on input-output relationships and does not contain preconceived ideas of the researcher.

The basic GMDH algorithm constructs a high-order Volterra-Kolmogorov-Gabor (VKG) polynomial of the form (1):

$$y = a_0 + \sum_{i=1}^{n} a_i x_i + \sum_{i=1}^{n}\sum_{j=1}^{n} a_{ij} x_i x_j + \sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{n} a_{ijk} x_i x_j x_k + \dots \tag{1}$$

that from n inputs $x_1, x_2, x_3, \dots, x_n$ is generated a single output y.

Ivakhnenko showed that the VKG series can be expressed as a network, that is cascade of of second order polynomials with only two input variables [2, 3].

1.1.1 GMDH Layers

Each layer in the network uses the nodes in the previous layer as inputs, first layer uses input data columns. Nodes that do not contribute to the solution and removed from the layer.

Because only n layer is needed for the creation of the $n+1$ layer, the whole notion of the network is sometimes omitted and replaced with sets of old and new variables, this makes scene for example in the classical Combinatorial GMDH algorithm, that creates new nodes from all the possible combination of the result pairs of the old nodes, this algorithm will be described in the next section.

1.1.2 GMDH Algorithm

To describe the algorithm, we need input data set with columns $x_1, x_2, x_3, \dots, x_n$ (inputs) and column y (result). Data should be split into two subsets; one for training and one for evaluation. The training set will be the first tsr rows from the total tr rows. Creation of the polynomial (1) will be described in the following steps:

1. Construction of the new variables - to construct new variables $x_1, x_2, x_3, \dots, x_{\binom{n}{2}}$. We take all $\binom{n}{2}$ combinations of the input pairs u, v from tsr rows of $x_1, x_2, x_3, \dots, x_n$ and find the best least squares polynomial, that best fits the results.

$$y = C_1 + uC_2 + vC_3 + u^2C_4 + v^2C_5 + uvC_6 \tag{2}$$

Now we use (2) to calculate the y (result) value for every input data row of every u, v input pair and store the results for that pair as z_m, where m is the index of that pair.

2. Selection of the new inputs - calculate least square error (se) between $z_1, \ldots, z_m$ obtained in the previous step and real result y, for every input data row as given in (3).

$$se = \sum_{i=1}^{tr} (y - z)^2 \tag{3}$$

If the result $se > M$ where M is predefined threshold, the pair is discarded. Find the smallest se, and if this se is greater than the smallest se obtained during the previous execution of this step, discard every z and go the the next step (if this is the first execution of this step, skip this part). All remaining results of z will be sorted by se and will replace inputs x and at then the algorithm will go back to the 1st step.

3. Selection of the result - first x column will have the smallest se and this result represent the result of the GMDH polynomial (1). To obtain the coefficient "$a_0, a_i, a_{ij}, a_{ijk}, \ldots$", of the polynomial (2) every input that was used in every iteration must be saved and evaluated.

1.1.3 Singular Value Decomposition and GMDH Coefficients

In the previous part it was mentioned, that coefficients $C_0, \ldots, C_6$ of the u, v input pair polynomial (2) are obtained by finding least squares polynomial, that best fits the results y. Singular Value Decomposition (SVD) is a method that can be used for solving most linear least square problems.

The method to obtain the coefficients, that will be described was proposed in[5].

To obtain the $C_0, \ldots, C_6$ with SVD, the following steps must be taken:

1. Create matrix $A = (1, u, v, u^2, v^2, uv)$ from u, v training data columns
2. Perform SVD on A: $A = U\Sigma V^*$
3. Invert every value in Σ
4. Get coefficients by calculating: $C = V^*\Sigma U^T Y$, where Y is array with results.

1.2 Differential Evolution

Differential Evolution (DE) algorithm introduced by Storn and Price [12] is a type of evolutionary algorithm (EA) and a complementary of Genetic Algorithm (GA).

The algorithm was originally designed to work with continuous variables. Onwubolu [6] introduced the forward/backward transformation techniques, that makes solving of any discrete or combinatorial problem possible.

1.2.1 DE Algorithm

GA uses biology inspired operations of crossover, mutation and selection on a generation to minimize an objective function over the course of successive generations [1]. DE as a GA also uses there operations, but in contrast to classical GAs uses floating-point values to represent the individuals instead of bit strings and uses arithmetics operations instead of logical during the mutation process.

The DE algorithm is composed of these steps:

1. Initialization of the population - Population must be initialized based on how many individuals we want in the population. This size will remain constant during the execution of the algorithm and is labeled NP.

   Individuals are initialized with random values, that are constrained only by the boundaries of the objective function parameters, that they represent. Number of objective function parameters in the individuals (dimension of the individual) will be called D.

2. Mutation - Mutation is used to create new trial population from the current generation, but does not need to be applied on every objective function parameter of the individual. If mutation is to be applied, it will be decided in the next step.

   Equation 4 is used for the mutation of selected objective function parameter of the selected individual:

   $$v_{j,i} = x_{j,r_1} + F \cdot (j, x_{r_2} - x_{j,r_3}) \tag{4}$$

   where v denotes a value in the trial population and x denotes a value in the current generation. $j = [1, D]$ is the index of the objective function parameter, $i = [1, NP]$ is index of the individual in the trial population and $r_1, r_2, r_3 \in [1, NP]$ are randomly selected integers, except $r_1 \neq r_2 \neq r_3 \neq i$, that represents indices of the individuals in the current generation. $F \in (0, 1]$ is a positive scale factor, that is typically less than 1 and remains constant during the execution of the algorithm.

   There exists more than one mutation strategy. This described strategy is usually called rand/1. For example another strategy is best/1, which is the same same as the rand/1, except the parameter r_1 is not a random integer, but index of the best individual from the current population.

3. Crossover - Not every objective function parameter in every individual in the trial population should be mutated. The selection process selects the objective function parameters that will be mutated.

   Two versions of the selection process exists, exponential and binomial:

Exponential : In this selection, a continuous section of the individual is mutated.

The following steps are applied to every individual in the trial population:

(a) Index of the fist objective function parameter in the section is randomly selected and mutation of this objective function parameter is performed.

(b) If random floating-point point value in range $[0, 1]$ is lower than crossover probability $CR \in [0, 1]$, which remains constant during the execution of the algorithm, the index is increased and mutation of objective function parameter with new index is performed and step 3a is repeated.
Note: if index is out of the dimension of the individual, it will be set to 1.

Binomial : In this selection, the objective function parameter to be mutated does not need to be continuous.

The following steps are applied to every individual in the trial population:

(a) Mutation is performed on randomly selected objective function parameter.

(b) For every other objective function parameter, if the random floating-point point value in range $[0, 1]$ is lower than the crossover probability $CR \in [0, 1]$, then the mutation of this objective function parameter is performed with (equation 4)

In both versions, all un-mutated objective function parameters of the individuals have the same values as the corresponding objective function parameters of the individuals in the current generation.

Both versions also use the same crossover probability $CR \in [0, 1]$ and will result in at least one mutated objective function parameter in every individual of the trial population.

4. Selection - Selection creates new population from the current one and the trial population.

   If the fitness of the individual of the trial population with index i is higher, then the fitness of the individual of the current generation with the same index i is replaced by the individual from the trial population. This procedure is performed for $i \in [1, D]$.

5. Stopping Criteria - steps 2, 3, 4 are repeated, until an individual with adequate fitness is found, or the specified number of iterations was completed.

As previously stated, there exists more mutation and selection strategies. The website of Price and Storn [10] contain implementation of 10 different working strategies of the DE in various programming languages and also contains advices regarding selection of NP, CR and F.

Informations about the performance of different strategies can be found for example in [4, 9].

1.3 Discrete Differential Evolution

As previously stated, Differential Evolutions works with floating-point values as objective function parameters, mainly because mutation and crossover operations of DE would would change integer values to real values, which would lead to infeasible solutions. The solution of this problem is a transformation of either population for mutation and crossover operations. A number of researchers decided to transform the population and leave DE unchanged.

One of such strategies advocates the forward and backward transformation that converts population between integers and real numbers. This new heuristic was termed Discrete Differential Evolution (DDE)[7].

Forward and Backward transformation from the[7] will now be described:

Forward Transformation Transformation from integer to real number is given by equation 5:

$$z = -1 + \frac{z' \times 500}{10^3 - 1} \tag{5}$$

where z is the new real number and z' is the original integer.

Backward Transformation Transformation from real number to integer is given by equation 6:

$$z = \frac{(1 + z') \times (10^3 - 1)}{500} \tag{6}$$

where z after proper rounding is the new integer and z' is the original real number.

In the DE algorithm, Forward transformation is performed on the whole population before the mutation and Backward transformation is performed on the newly created population.

2 Aim and Objectives

The aim of the thesis was to create an implementation of a Hybrid GMDH, with Singular Value Decomposition (SVD) as a regression analysis for the calculation of the coefficients of the nodes. As there exists more than one hybrid version of GMDH, we wanted our implementation to have good extendability and modularity options, therefore GMDH version that obtains the whole network structure from an individual, which implies that any algorithm which works with individuals can be used for the creation of the best network structure was selected. This GMDH version also gives more possibilities, when it comes to memory usage optimization and parallelization, because it is composed of two almost independent algorithms (GMDH itself and algorithm for creation of the best network structure) and both can be optimized and parallelized in different ways. As specified in the thesis assignment, SVD is used for the coefficients calculation, but was implemented in such a way that makes it easily replaced with other method to obtain the coefficients.

This implementation uses Discrete Differential Evolution as an algorithm for the creation of the best network structure, but as mentioned earlier, it would be relatively easy to replace it with another algorithm.

This framework was created to work efficiently even with larger Network Models (over 20 layers) Soft limit is 30 layers (individual representing a network with 30 layers takes 4GB of memory and contains up to $1,073,741,823$ nodes).

This application is cross-platform, it is compilable on Microsoft Windows, Mac OS X, and most of the GNU/Linux distributions. (It also should be compilable on most of the other UNIX and UNIX like platform, but this was not tested). Target platforms are personal computers, laptops and high performance computers.

GCC is the recommended compiler, as with this compiler you can choose to use platform specific thread implementation (windows threads for Microsoft Windows or POSIX threads for Mac OS X and GNU/Linux) or Open MP. On other compilers it is recommended to use Open MP variant, because version with platform specific thread implementations uses some GCC specific operations.

Verification of our application was carried our by testing it on data sets: [13, 8, 11].

3 Implementation

3.1 DE-GMDH

In this research, Differential Evolution takes care of the search for the best Network Model and the whole Network Model is defined by the individual in the population of the DE.

This concept changes the network creation paradigm. In the original GMDH, fitness of every node is evaluated during the network creation and the result of the network creation is the final network. In our algorithm, each individual has its own network, and the network creation and evaluation can be separated and the final network is the network of the individual with the best fitness. Same concept can be used on other GAs [5].

In our DE-GMDH, the following **rules** hold true:

1. Nodes are the same as the ones in the standard GMDH (2 inputs, 1 result, coefficients, ...).

   Theoretically, the order of the inputs does not matter, because u, v and u, v pairs should end up with the same coefficient, except C_2 and C_3 being swapped, but because of the floating point errors, this does not hold and will be addressed in the next section.

2. No nodes will be removed based on $se > M$, therefore M parameter no longer exists.

3. Result of every node must be used.

4. Layer must have at least 1 node.

5. Last network layer will have only 1 node, the result node.

6. Node can use results of the nodes from all previous layer as its inputs.

   This rule was added to expand the number of possible network structures, by creating nodes, that will be used as inputs by the nodes in higher than next layer, which would not be possible in the original GMDH.

7. Structure of the whole network will be presented as individuals in the population.

8. Columns with input data are considered to be results of the nodes of the 0 layer. This rule exists sorely to make the explanation easier.

From these rules, we can make following observations:

1. Number of the nodes in the last layer is 1 as stated in rule 5.

2. Number of nodes in the n layer, where n is not the number of the last layer, is at most two times more than the number of nodes in $n + 1$ layer as deduced from the rules 1, 6 and 3.

It may be assumed that rule 6 gives rise to the possibility of having for example a network with 5, 2 and 1 nodes in each layer, where in the first 4 nodes of the 1st layer are used by the nodes in the 2nd layer and the remaining 1 node is used by the node in the 3rd layer, but such situation can not occur, because one node from the 2nd layer would be unused, therefore it would have to be removed along with its inputs and you would end up with network with 3, 2 and 1 node. This implies, that if some node uses result of the node that is more than 1 layer down, it will always lead to less nodes in the network.

3. Number of nodes in the 1st layer is $2^{(lc-1)}$ where lc is number of positive network layers as deduced from observation 2.

4. Maximal number of nodes in the network is $2^{lc} - 1$ where lc is number of positive network layers as deduced from observation 2 and rule 5.

As stated in rule 6, we need to find a way to present the GMDH network as an individual in the population.

If we change rule 6 to having nodes in layer n use the results of the nodes in layer $n-1$ as their input, then observations 4 and 3 will give the exact number of nodes, instead of maximal. It is assumed, that there are no constraints when in comes to which node uses which input, we realize, that if we freely choose inputs of the nodes in the 1st layer and inputs for all the other nodes in the other layer, we can obtain all possible networks.

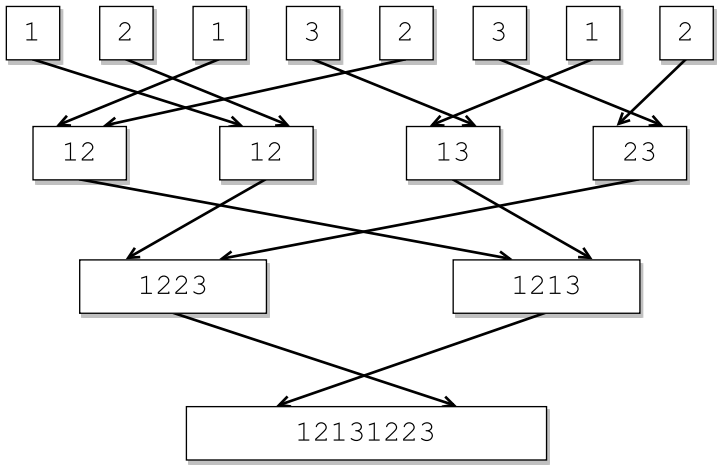

Figure 1: Network of DE GMDH with no structure constraints

Figure 1 shows the network with no structural constraints and figure 2 shows the network with predefined structure of every layer except the 0 layer.

Name of the node is the combination of the names of its inputs, nodes with 1 letter name are not nodes, but input data columns (nodes of the 0 layer).

It is clear, that result nodes of the two networks are the same, therefore the networks themselves must be the same. The only difference is the order in which the nodes were drawn.

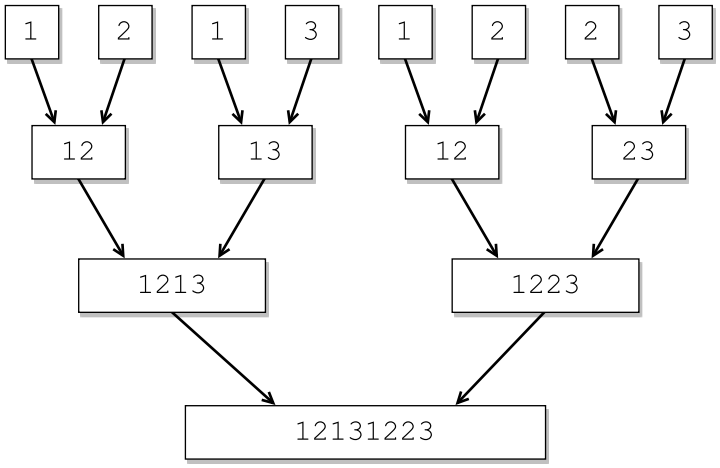

Figure 2: Network of DE GMDH with structure constraints in every layer, except the 1st one

This demonstrates that the network with a predefined structure in every layer except the 1st one is just a re-ordered version of any other network.

With having this knowledge, the creation of the population that represents such a network is trivial, as such a population is simply the sequence of the nodes in the 0 layer of the network. Because nodes in the 0 layer are input data columns, we know their range and we also know the number of nodes in the 0 layer is 2^{lc} where lc is the number of positive network layers.

Therefore, we can reiterate to the original rule 6. We will use the assumption that the node cannot have the same inputs. Nodes having the same input are permitted, but the nodes will not do any calculation and will simply use one of its inputs as the result. Therefore, such nodes are redundant, however they keep the network structure the same. Figures 3 and 4 demonstrate this concept.

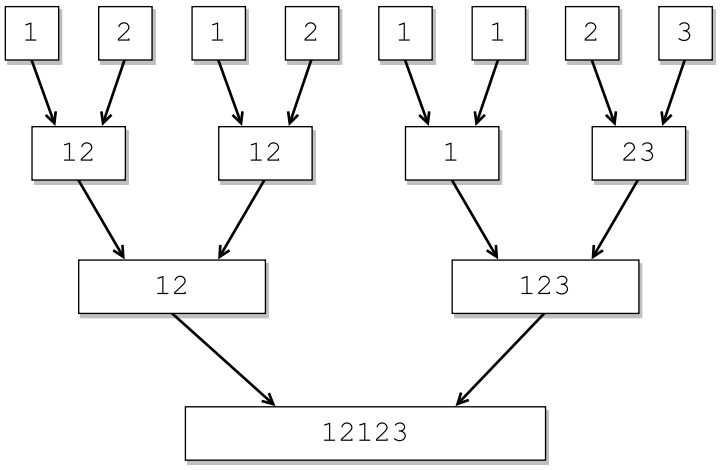

Figure 3: Network of DE GMDH with nodes that do nothing

The preceding description has described the generation of the network as an individual in the population. The fitness evaluation of the individual is the square error of the network.

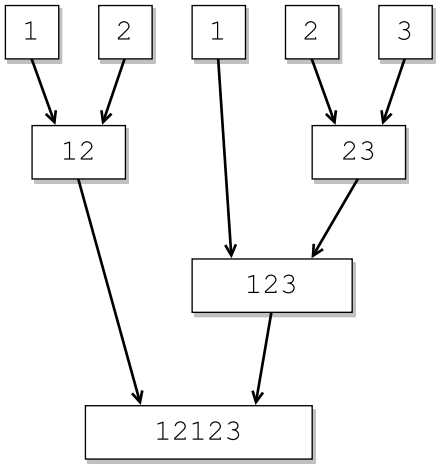

Figure 4: Same network as the one in 3, but without the useless nodes

As this point Discrete Differential Evolution (DDE) is applied to the population (values in the individuals are indices of the input data columns, therefore integers), until some individual has the sufficient fitness or the population is evaluated.

3.2 DE-GMDH Network Structure Optimization

From the previous section, it is obvious that the network created from the specification of all the inputs to all the nodes in the 1st layer is not optimized. It may contain nodes with the same inputs that will obviously have the same result and therefore, there is no need to calculate the result of more than one of them and to simply reuse it. As already mentioned, the order of inputs actually matters.

In fact, most of the Network Models will contain redundant nodes (redundant node is a node that has the same inputs as some other node), because the structure of a network with no redundant nodes is severely limited, number of nodes in the 1st layer of such network is at most $\binom{idcc}{2} + idcc$ where $idcc$ is the number of input data columns and any next layer would have to have at most half of the nodes that were in the previous layer. Therefore, redundant nodes are in fact necessary part of any network.

3.2.1 Network Model

Before we start with any optimization, it is worth mentioning, that we can not store the optimized network back into the individual, because it would destroy the properties of the individual that are needed for the DDE.

Therefore, we need to define a structure to hold the resulting network. It is obvious that different types of optimization will require different attributes in the Network Model, to make the explanation easier, complete model with all the needed attributes will be shown and then the attributes themselves will be described in the following sections.

The Network Model will contain:

- Number of network layers
- Number of nodes in each layer
- Information about each node in each layer

Information about each node will contain:

- Index and layer of each input
- Coefficients
- Number of nodes, that use the result of this node as their input (Number of Users)

3.2.2 Non-redundant Ordered network

As the name suggest, this network will contain no redundant nodes and the nodes will be ordered. To be able to find redundant nodes, we need to store the non-redundant nodes in a container that stores only non-redundant items and has a relatively fast addition and search. Items should also be easily accessible in an ordered form. Binary tree fulfills these conditions, therefore in the actual implementation, R-B tree is used, but obviously the basic algorithm does not require it.

During the optimization we need to store:

- Network Model of the optimized network.
- Binary tree (or other appropriate container) with non-redundant nodes of the currently processed layer. This container will be called the **node buffer**.

  We store layers and indices of the inputs of the node, that we want to add to this container, the node itself is not stored. Therefore, when we talk about checking if node is in this container or adding it to it, we mean the set of the indices and layers of the node inputs.

  Unique index of the node also need to be stored. It is obtained simply by storing number of nodes in the contains, before the node was added (1st added node will have index 0, 2nd 1, 3rd 2, ...).

- Re-indexing array. This array translates the index of the node in the **node buffer** to the index of the node in the Network Model. This array is required because nodes in the Network Model are ordered, therefore we do not know their index until all the nodes of the layer are processed. This array must exist for each layer, because nodes can use nodes from any previous layer as their input. This array will be called **reindex 2D array**.
- An array, that currently holds the processed unoptimized layer and after processing of that layer will hold the next unoptimized layer, that will be processed again in the following iteration. This array will be called **layer buffer**.

Because the size of the layers in the unoptimized network are fixed and the size of the next layer is always half the size of the current layer, size of this array will only decrease.

Each node in this array contains layer of that node in the Network Model and index. If the layer is -1, then the index refers to the index of input data column, otherwise it refers to index of the element in the **reindex 2D array** of that layer, which holds the index of the node in the Network Model. Therefore, when Index of the node is mentioned, it is assumed that it was properly re-indexed.

- Index of the layer in the Network Model that will be created from the **node buffer**, because the number of layers of the optimized and unoptimized network may differ. For example, the node in the last layer of the unoptimized network may have the same inputs, therefore, that node does nothing and at least half of such a network is redundant. Another example: network $1, 1, 2, 2$ is an unoptimized network with 2 layers, but nodes in the 1st layer do nothing, so after optimization, it will be network $1, 2$ with just one layer. This variable will be called the **layer index**.

Notes:

- Inputs of the nodes in the 0 layer are input data columns.
- Inputs of the nodes in n layer, where n is positive are the nodes from the $n-1$ layer, that are stored in **layer buffer** or input data columns, if layers of that node is set to -1.
- If inputs of some node are compared or ordered, we obviously talk about the layer and index values of the two nodes that are the inputs of that node.

The following is the procedure of the processing of the 0 layer:

1. Set **layer index** to 0.
2. For each input pair in the individual, check if the inputs differ.
    - If the inputs are the same, it is assumed that this node does nothing and therefore there is no reason to have it in the optimized network. In the **layer buffer**, set layer of the node that corresponds to this pair to -1 and index to the value of the pair.
    - If the inputs differ, order them and in the **layer buffer** set layer of the node that corresponds to this pair to 0 (1st layer) and index to its index in **node buffer**. If this node is not in the **node buffer**, then add it.
3. At this point, the **node buffer** holds all nodes in the 1st layer of the optimized network and **layer buffer** holds the indices and layers of the nodes in the Network Model (after re-indexing) for every node in the unoptimized 1st network layer.

    If **node buffer** contains at least one node:

- Copy ordered **node buffer** to the 1st layer of the Network Model. and delete the **node buffer**. (in case of binary trees, simply in-order tree traversal and delete)
- Set **reindex 2D array** for the 1st network layer.

  Get index of the node from the **node buffer** and set the value of the element with that index to the index of that node in the 1st layer of the Network Model.

  Example: If node with **node buffer** index 5 has index 77 in the 1st layer of the Network Model, then **reindex 2D array** element in the same layer with index 5 will have value 77.

  Note: This action is obviously carried out during the creation of the 1st layer of the network Model, because after that, the **node buffer** will be deleted.
- Increase the **layer index**.

Processing the n layer where n is positive:

1. For each node pair in the **layer buffer**, check if the inputs differ.

   - If the inputs are the same, then the node, which uses these 2 nodes as inputs does nothing and therefore there is no reason to have it in the optimized network. In the **layer buffer**, set layer and index of that node to the layer and index of one of its input nodes.
   - If the inputs differ, re-index and order them and in the **layer buffer** set layer of the node, which uses these 2 nodes as inputs to **layer index** and index to its index in the **node buffer**. If this node is not in the **node buffer**, then add it.

2. Now the **node buffer** holds all nodes in the n layer of the optimized network and **layer buffer** holds the indices and layers of the nodes in the Network Model (after re-indexing) for every node in the unoptimized $n + 1$ network layer.

   If **node buffer** contains at least one node:

   - Copy ordered **node buffer** to the **layer index** layer of the Network Model and delete the **node buffer**. (in case of binary tree, simple in-order tree traversal and delete)
   - Set **reindex 2D array** for the **layer index** network layer.

     Get index of the node from the **node buffer** and set the value of the element with that index to the index of that node in the **layer index** layer of the Network Model.

     Example: If node with **node buffer** index 5 has index 77 in the **layer index** layer of the Network Model, then **reindex 2D array** element in the same layer with index 5 will have value 77.

     Note: This action is obviously carried out during the creation of the 1st layer of the network Model, because after that the **node buffer** will be deleted.

- Increase **layer index**.

As you may have noticed, individuals like $1, 1$ will result in no optimized network, which is not an issue, simply because the network that uses one of its inputs as the result and does no further computation.

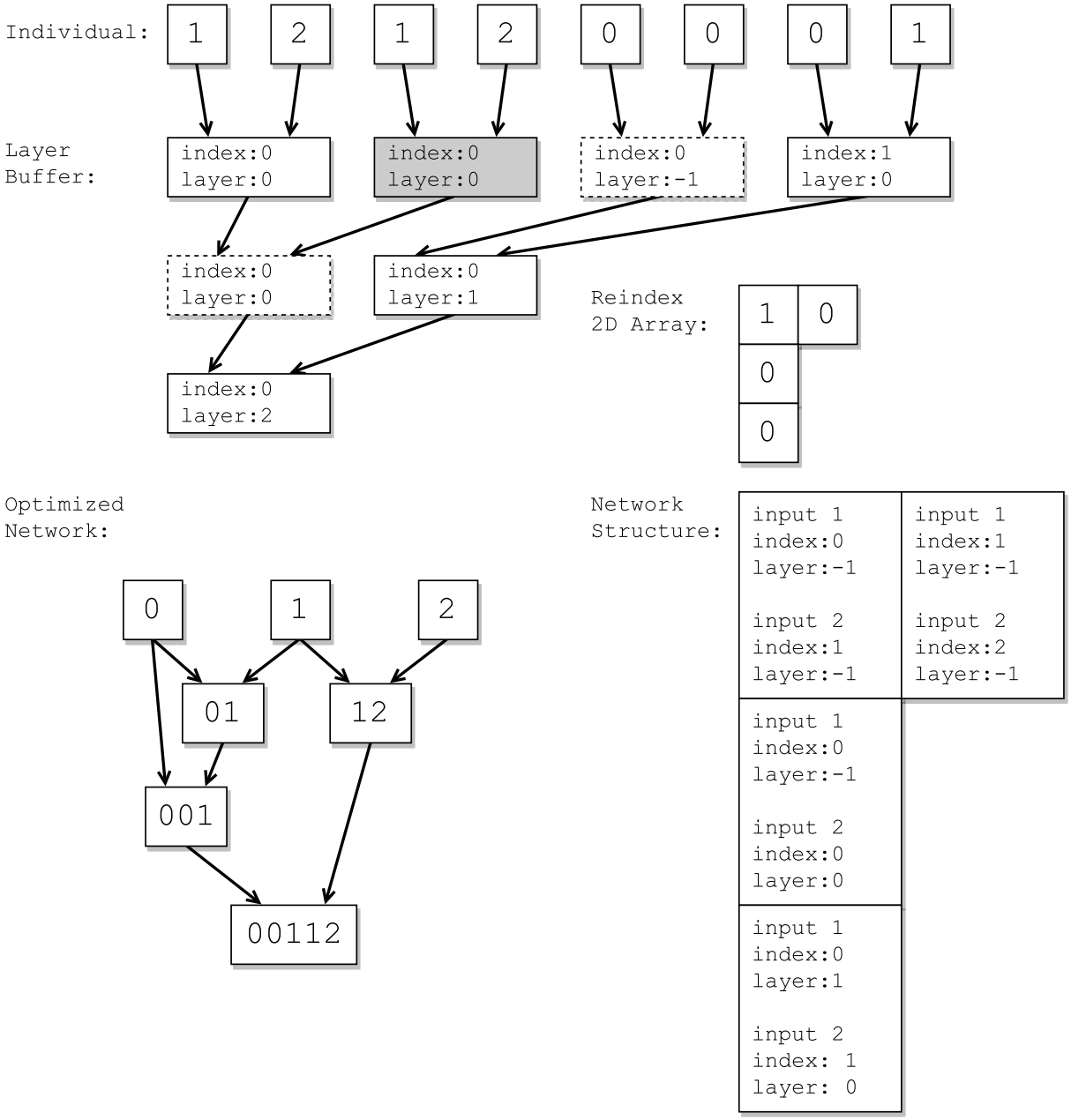

Figure 5: Example of the network redundancy and order optimization

Figure 5 illustrates the entire process. Individual which is to be optimized has both redundant nodes and wrong input order. Individual and then **layer buffer** are always processed in from 1st to last element order. Ascending order is used during the ordering, with nodes first ordered by

input indices, then by input layers. Elements of the **layer buffer**, that are drawn with dashed line are nodes, that do nothing. Elements with gray background are redundant nodes.

Nodes in the **node buffer** will be described as 5 numbers in the format $i1, l1, i2, l2 - ind$ where:

i1 - Index of the 1st input.

l1 - Layer of the 1st input.

i2 - Index of the 2nd input.

l2 - Layer of the 2nd input.

ind - Unique index of the node.

Just a reminder, that the **layer buffer** is a single dimension array, that decreases in size by half in each layer processing.

The following procedures are as follows:

1. 1st $1, 2$ input pair was selected, and this pair corresponds to the 1st node in the **layer buffer**.

   Inputs differed and already have the correct order. Set layer of the node to 0, because it will be in the 1st layer of the optimized network. Node is not in the **node buffer**, therefore it is added and receives index 0 (1st node in the **node buffer**). Set index of the node to 0.

2. 2nd $1, 2$ input pair was selected, and this pair corresponds to the 2nd node in the **layer buffer**.

   Inputs differed and already have the correct order. Set layer of the node to 0, because it will be in the 1st layer of the optimized network. Node already is in the **node buffer** with index 0, set index of this node to that index. (Node has gray background, because it is redundant.)

3. Input pair $0, 0$ was selected, and this pair corresponds to the 3rd node in the **layer buffer**.

   Inputs are the same, set layer of the node to -1 and index to the the value of one of its inputs. (Node is drawn with dashed line, because it does nothing.)

4. Input pair $0, 1$ was selected, and this pair corresponds to the 1st node in the **layer buffer**.

   Inputs differed and already have the correct order. Set layer of the node to 0, because it will be in the 1st layer of the optimized network. Node is not in the **node buffer**, therefore it is added and receives index 1 (2nd node in the **node buffer**). Set index of the node to 0.

5. Now, the **node buffer** holds 2 nodes: $1, -1, 2, -1-0$ and $0, -1, 1, -1-1$.

   If we order them, we will get: $0, -1, 1, -1-1$, $1, -1, 2, -1-0$ and that is the 1st layer of the Network Model.

6. Node $0, -1, 1, -1-1$ has index 0 in the 1st layer of the Network Model and unique index 1, therefore Element in the 1st layer of the **reindex 2D array** with index 1 will have value 0.

7. Node $1, -1, 2, -1-0$ has index 1 in the 1st layer of the Network Model and unique index 0, therefore Element in the 1st layer of the **reindex 2D array** with index 0 will have value 1.

8. Processing of the 1st layer is completed, clear the **node buffer**. Proceed to the 2nd layer.

9. 1st pair in the **layer buffer** was selected; this pair corresponds to the 1st node in the **layer buffer** for the new layer (same buffer).

   Inputs are the same, so there is no need for the re-indexing. Set index and layer of the node to Index and layer of one of its inputs. (Node is drawn with dashed line, because it does nothing.)

10. 2nd pair in the **layer buffer** was selected; this pair corresponds to the 2nd node in the **layer buffer** for the new layer (same buffer).

    Inputs are different, therefore re-index them.

    1st input has layer -1, therefore no re-indexing index of this input is 0.

    2nd input has layer 0 and index 1, element in the layer 0 of the **reindex 2D array** with index 1 has value 0, therefore index of this input is 0.

    Now, we have inputs $0, -1$ and $0, 0$, therefore the order of the inputs is correct.

    Node is not in the **node buffer**, therefore it is added and receives index 0 (1st node in the **node buffer**). Index of the node was set to 0.

11. Now **node buffer** holds 1 node: $0, 0, 1, -1-0$. This is 2nd layer of the Network Model.

12. Node $0, 0, 1, -1-0$ has index 0 in the 2nd layer of the Network Model and unique index 0, therefore Element in the 2nd layer of the **reindex 2D array** with index 0 will have value 0.

13. Processing of the 2nd layer is complete, therefore clear the **node buffer**. Proceed to the final layer.

14. 1st and only pair in the **layer buffer** was selected; this pair will create the last node of the network. (Only node in the last layer)

Inputs are different, re-index them.

1st input has layer 0 and index 0, element in the layer 0 of the **reindex 2D array** with index 0 has value 1, therefore index of this input is 1.

2nd input has layer 1 and index 0, element in the layer 1 of the **reindex 2D array** with index 0 has value 0, therefore index of this input is 0.

Now we have inputs $1, 0$ and $0, 1$, after ordering we get $0, 1$ and $1, 0$.

Node is not in the **node buffer**, therefore it is added and receives index 0 (1st node in the **node buffer**). Index of the node was set to 0.

15. Now the **node buffer** holds 1 node: $0, 1, 1, 0 - 0$. This is 3rd and final layer of the Network Model.

16. We know this is the last network layer, therefore we do not need to create re-indexing records for the next layer, but in the actual implementation, it might be easier to generate it, as it will hold just 1 index, so there is very little overhead.

17. Optimization was completed, delete everything except Network Model and individual.

3.3 Memory Management

In this section, we will discuss the memory management during the evaluation of the network, because this calculations usually requires the most memory and can be carried out in different orders, which impacts memory usage. On the other hand, creation of the ordered non-redundant network probably can be carried out differently, but most of the memory usage is unavoidable (memory used by the individual, Network Model) or necessary for speed of the processing (binary tree instead of the search in the individual, re-indexing array instead of rewriting every single value in the layer)

Generally speaking, during the evaluation of the network, most of the memory will be used to hold the results of the nodes. Size of this memory depends on the number of the input data rows, so if we have six input data rows, then the memory, which is used to hold the result of the node is the same as the memory needed to store the coefficients. But in the general cases, input data has at least tenths of rows. Therefore, the memory management will mostly be about the maximal number of the stored results during the network evaluation.

Evaluation of the network represent these operations:

Note: As previously stated, Network Models are generated from the individuals of the population, therefore when we talk about performing one of these operations on the individual, we mean performing on of these operation on the Network Model, that was generated from that individual.

Fitness evaluation The most common operation, that is used to evaluate the fitness of the network.

Because this operation must be carried out on each individual in each the generation, until the best individual is found, we can lower memory usage by discarding the calculated coefficients of the node after its result is calculated. Doing so will make it impossible to obtain the GMDH polynomial (1) of the network, therefore the best Network Model will have to be evaluated again.

Coefficients Calculation This operation is performed to obtain the coefficients of every node in the Network Model.

It is essentially **Fitness evaluation**, without coefficients discarding.

It may be assumed that **Fitness evaluation** will not save any memory, because this operation must be performed on the best individual, that may be true, if the best individual is also the individual with the higher memory usage and one individual is evaluated after the other. If the best individual is individual with for example only half of the memory usage compared to the individual with the highest memory usage, then using only **Fitness evaluation** on the individual with the highest memory usage will help. Another example may be parallel **Fitness evaluation** of more individuals, in that case it is better to lower the maximal memory usage of each individual.

It may be also assumed that by using **Fitness evaluation** and then this operation one more network evaluation is added and the calculation time is increased, that is true, but most of the time hundreds of **Fitness evaluations** are carried out before the best individual is found, which means, that only a fraction of the total time will be used for the final evaluation. If the network are small and use little memory, evaluation time is also very short, therefore these is little point in not using

Result Calculation This operation is performed to obtain the result of the network, network must be fully defined (coefficients must be known).

This operation is used to calculate the results for any additional input data.

You may have noticed that the Network Model, that was generated during the network optimization did not define the number of nodes that use the result of this node as their input. That is because this parameter of the node in the optimized network is used for memory usage management. It is obvious that the result of the node is not needed after it was used by all the nodes, that require it. Therefore, we will need to obtain the number of users of each node.

Number of users of each node can be determined after the Network Model is defined, but it is more efficient to do it during its creation, more specifically during the addition of the new node to the Network Model, perform:

1. Set its **Number of Users** to 0.

2. Find nodes in the Network Model, that this node uses as inputs and increase their **Number of Users** by 1. (if node uses input data columns as inputs, do nothing)

Discarding the result of the node after is no longer needed is not all we can do. In most cases allocation of the memory for the result of the node takes time, therefore if this node is the last used of at least one of the the result of its input it can reuse memory of such result for its own result. In such case we only need to discard result of the node, if its last user already reuses memory of its other input and allocation of the memory happens only if no input memory can be reused (this typically happens only in the lower layers of the network, where input data column are used as inputs and nodes have high **Number of Users**).

3.4 Calculation Order

Order in which the nodes are processed influences the memory usage, as is obvious, that the order cannot be random, because before processing a node, its inputs must be defined, and input of a node can be another node.

Two significantly different orders will be described:

Layer-by-Layer Typical order for processing network with n layers, where input of any node in the n layer is from the lower layers (bottom up).

This order is used in the original GMDH algorithm and during the creation of the Network Model from the individual. It also it the only reasonable order, that these algorithms can use, because it can work with incomplete network structure.

Recursive Recursive top down processing of the network.

This order requires complete network structure to work, therefore can be performed only on Network Model

Regardless of the approach, the **Number of Users** is used in the same way, when processing the node, we decrease the **Number of Users** of each input node (if input is from the input data, we do nothing).

- If **Number of Users** of only one input node reaches zero, result of this node will use the memory, that was used to store the result of that node.
- If **Number of Users** of both input nodes reaches zero, result of this node will use the memory, that was used to store the result of the first input node and memory, that holds the result of the second input node will be discarded.
- If **Number of Users** of both input nodes is non zero, new memory will be allocated to hold the result of this node.

3.4.1 Layer-by-Layer Calculation Order

This processing order is well known and very simple; pseudocode is in Listing 1.

```
main()
{
   for every network layer
   {
      for every node in the layer
      {
         process node; // calculate coefficients, result ...
      }
   }
}
```

Listing 1: Pseudocode: Layer-by-Layer network Processing

Peak number of stored results is influenced mainly by the number of the nodes in each layer, because we have to process every node in each layer before moving on to the next one. Therefore, after processing the last node in the layer, result of every node in that layer must be stored and also results of the nodes from the previous layers, that were not used in this layer.

3.5 Recursive

Recursive processing order is simple a top-down recursion; pseudocode is in Listing 2.

```
main()
{
   processNode(last node); // only node in the last network layer
}

processNode(node)
{
   if (input1 of the node is not processed)
   {
     processNode(node->input1);
   }

   if (input2 of the node is not processed)
   {
     processNode(node->input2);
   }
```

```
        process node; //calculate coefficients, result ...
    }
```

Listing 2: Pseudocode: Recursion network Processing

This recursion is considered because if the result of a node was used by exactly one node, then maximal number of results in the memory would be the same as the number of layers of such network.

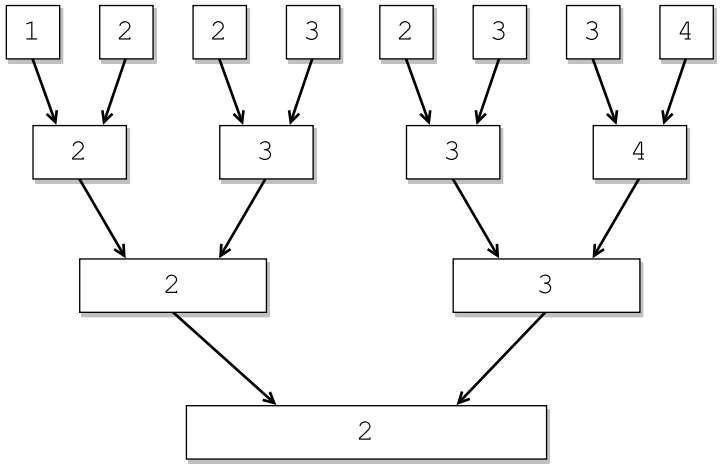

Figure 6: Recursive order results in memory - restricted network

This number may sound unrealistic, therefore figure 6 illustrates the concept on which it was obtained. 1st input of the node is an arrow, that is on the left top size of the node. Number in each node represents the number of the results in the memory during the processing of that node, however, please note that the nodes in the 1st layer use input data columns as their inputs, therefore during their processing they require only one result in the memory (newly allocated memory for their result). It can be easily observed that the number of the inputs in the memory during the node calculation is number of the results needed to store the results of its inputs (1 for the nodes in the 1st layer, 2 of very other node) plus number of the 1st input nodes of the nodes, that are in the "path" from this node to the last node in the network.

This obviously holds true only in that very restricted network, where no nodes share the results. In general, network number of already processed nodes, whose results will be used in nodes that were not processed yet is added to that number. Results of such nodes must be used by more than 1 node, and it would be possible to recalculate their results, but that would make no sense, because the whole non-redundant optimization would be lost.

3.6 Parallelization

In this section, we will discuss only thread based parallelization. Parallelization can be utilized in the original GMDH algorithm, but is restricted by the fact, that the original GMDH basically uses Layer-by-Layer order of node evaluation, which means, that only that order can be

parallelized, our DE-GMDH is separated to DDE, Network Model creation and evaluation of the network. Each of these parts can be parallelized and has different properties if parallelized.

First we need to decide, what properties should the parallelization have, because parallelization of the DDE (processing more individuals at the same time, each by 1 thread) has different properties than parallelization of the Network Model creation and evaluation of the network (processing of the 1 individual, by more threads)

3.6.1 Parallelization of Network Model creation

Because Network Model creation uses Layer-by-Layer order parallelization is straight forward:

- Split processing of each layer between the threads, by splitting the array with nodes to sub-arrays.
- Each thread will have its own **node buffer**, but unique index, that node receives, will be globally unique (synchronized between the threads)
- **reindex 2D array** will be shared.
- Instead of simply copying ordered **node buffer** to the **layer index** layer of the Network Model, we need to remove duplicates between different **node buffers**. The algorithm to do that is basically the Merge Sort.

Parallelization of the Network Model creation adds memory overhead, because **node buffers** may contain same nodes, therefore **reindex 2D array** must be bigger. If we combine this with the poor results on the small network and the fact, that time that it takes on the small network is almost insignificant, we would recommended to use it only on the large networks (20 layers plus).

3.6.2 Parallelization of Layer-by-Layer network evaluation

As previously stated, Layer-by-Layer parallelization is straight forward:

- Split processing of each layer between the threads, by splitting the array with nodes to sub-arrays.
- Process all sub-arrays.

Memory usage should be roughly the same as the one of the unparameterized Layer-by-Layer processing, because memory usage can be higher, only if two or more threads are using the same node as input during the evaluation and after both of the evaluation are finished, that input node will have no more usages left, in this case both thread will have to allocate new memory for the result and after the evaluations are completed the memory of that input node will be

deallocated (reuse of the memory was not possible). This specific situation raises the possible peak number of result just by 1.

This situation is by its nature rare and its number of simultaneous occurrences is restricted by the number of threads, therefore only half the thread count of these situation can occur at once.

3.6.3 Parallelization of Recursive network evaluation

Recursive calculation order parallelization is not very intuitive. We call it **Wait and Work** parallelization.

The name describes the state in which one thread waits for the result of the node, that will be calculated by another thread and because it is idle while it waits, it will accept work from other threads.

Nodes are still processed recursively, but the processing function (`processNode`) is more complex:

- If both of the node's inputs are processed, then this node will be processed.
- If one of the node's inputs is being processed by another thread and the other is processed, **Wait and Work** for the unprocessed input will be performed, and then this node will be processed.
- If one of the node's inputs is processed and the other is unprocessed, then the unprocessed input will be processed, and then this node will be processed.
- If one of the node's inputs is being processed by another thread and the other input is unprocessed, then the unprocessed input will be processed, the **Wait and Work** for the other input will be performed, and finally this node will be processed.
- If both of the inputs are being processed, **Wait and Work** will be performed for both inputs, and then this node will be processed.
- If both of the inputs are unprocessed and there is no thread that can accept work, the first input will be processed, then these rules will be applied on this node gain. (only the first 3 states of the node will be possible)
- If both of the inputs are unprocessed and there is some thread that will accept work, that thread will process the 1st input and then will process the 2nd input, upon which these rules will be applied on this node again. (only the first 2 states of the node will be possible)

Main thread will start processing of the result node, every other thread will `waitForWork`. Please note, that `waitForWork` is not the same as **Wait and Work**, because in **Wait and Work**, threads wait for some node to be processed and also for the work, in `waitForWork` thread waits only for the work.

It is assumed, that thread that waits for example for a node in the 3rd layer may receive work on a node in the 10th layer, which creates a possibility of a deadlock. Because let us assume that this thread is processing node A and some other thread is processing one of the input of this node, this thread will go into **Wait and Work** and will receive work on a node B, that has node A as input and deadlock will occur, because this thread will be in the **Wait and Work** in the node B waiting for the node A to be processed, but that will never happen, because node B must be processed before **Wait and Work** on the node A ends.

Figure 7, shown an example of a network, that may result in a deadlock. We will describe a evaluation of this network, that will lead to the deadlock.

- Three threads are used during the network processing, `T0` , `T1`, `T2`, indices in their names also indicate the order in which they were created.
  - Nodes that thread `T0` is responsible for have the white background, and results of such nodes are connected with the other nodes with black arrows.
  - Nodes that thread `T1` is responsible for have light gray background, and results of such nodes are connected with the other nodes with black dashed arrows.
  - Nodes that thread `T2` is responsible for have dark gray background, and results of such nodes are connected with the other nodes with dark gray arrows.
- Processing of any node will take exactly 1 time unit. This rule is not realistic and is the main reason as to why in the real implementation the deadlock may not always happen.
- Name of the node is composed of the time, when the processing of that node will be finished and the thread that is responsible for the processing of that node. Because deadlock will occur every node that will never be processed, `NEVER` instead of time is used.

Important moments in the network evaluation can be described as follows:

1. At time 1.1, thread `T1` finishes the processing of the node `1.1 - T1` and enters **Wait and Work** for the node `3 - T0`.
2. At time 1.1, thread `T0` already started with the processing of the node `2 - T0` and therefore there is no work for any other thread.
3. At time 1.2, thread `T2` finished the processing of the node `1.2 - T2` and moves to the node that is drawn with dashed line, where if finds out that this node has 2 unprocessed inputs and because thread `T1` is in the **Wait and Work**, assigns the 1st input to it, and starts processing of the node `2.2 - T2`.
4. Thread `T1` receives the work from thread `T2`, node `1.1 - T1` which was already processed and node `NEVER - T1` that is drawn with dashed line is being processed, therefore thread `T1` enters **Wait and Work** for that node and deadlock occurs, because this node belong to this thread.

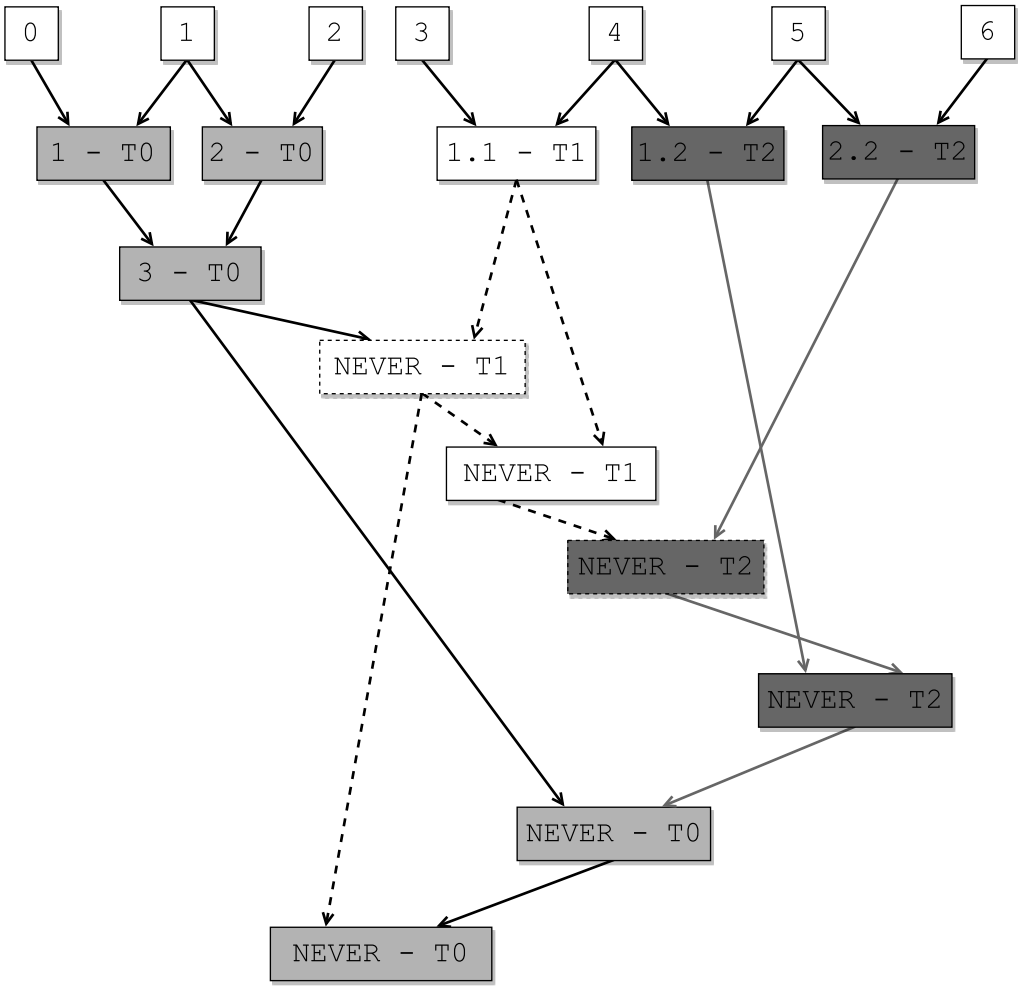

Figure 7: Multi thread Recursive Network Evaluation - Network with possible deadlock problem

This possible deadlock can be prevented, by allowing a thread in **Wait and Work** to work only on the nodes in the same or lower layer, than the layer of the node on which the thread waits. If the thread is in `waitForWork`, do not perform this check, because thread is not waiting for any node, therefore deadlock is not possible.

It may be assumed, that by allowing a thread to work only on the nodes that are being processed by the thread which is the cause of the **Wait and Work**, which may be more of a comprehensible idea, based on the simple notion of a thread helping the thread that it has to wait for, but this rule is more complex to implement, because you should also allow such thread to work on the nodes that are being processed by a thread that is processing the **Wait and Work** node. And this rule is also more restrictive, therefore a thread may have to wait instead of working on some other node in the same layer.

In the implementation, it is better to implement this anti-deadlock rule in the `processNode` function, by not giving such work to the thread in the first place, because implementation in the **Wait and Work** would require a system for rejection of the work, which would greatly

complicated things.

Pseudocode in Listing 3 illustrates this parallelization rules.

```
main()
{
   for N threads
      StartThread(waitForWork());

   // main thread
   mark main thread as thread that is working;
   processNode(last node); // only node in the last network layer

   for N threads
      TerminateThread();
}

waitForWork()
{
   mark this thread as a thread, that can perform some work;

   while(this thread exists)
   {
      if(there is some work for this thread)
     {
         mark this thread as a thread that is working;
         processNode(parameters received from other thread);

         mark this thread as a thread that can perform some work;
     }
   }
}


waitAndWork(node)
{
   mark this thread as a thread that can perform some work;

   while(node is not processed)
   {
      if(there is some work for this thread)
```

```
      {
          mark this thread as a thread that is working;
          processNode(parameters received from other thread);

          mark this thread as a thread, that can perform some work;
      }
    }

    mark this thread as a thread that is working;

    if(there is some work for this thread)
       processNode(parameters received from other thread); // in most
            of the real implementations, this step will be necessary,
           because some thread may give this thread work after the
           node was processed, but this thread still was not marked as
            a thread that is working.
}


processNode(node)
{
    if (input1 of the node is not processed and input2 is not
        unprocessed)
    {
       mark input1 as being processed;
       processNode(node->input1);

       if (input2 is being processed)
         waitAndWork(input2);

    }
    else if (input2 of the node is not processed and input1 is not
        unprocessed)
    {
       mark input2 as being processed;
       processNode(node->input2);

       if (input1 is being processed)
         waitAndWork(input1);
```

```
    }
    else if (input1 of the node is not processed and input2 of the
        node is not processed)
    {
       mark input1 as being processed;
       mark input2 as being processed;

      if(some thread can perform some work) // do not forget to check
          if the layer of input1 is lower or equal to the layer of
          the node on which the thread that can perform some work
          waits
       {
         tell that thread to process input1;
          processNode(node->input2);

          if (input1 is being processed)
             waitAndWork(input1);

       }
       else
       {
          mark input2 as not being processed; // if you don't do this
              you may get into deadlock
          processNode(node->input1);

          if (input2 is being processed)
            waitAndWork(input2);
          else if (input2 is not processed)
          {
             mark input2 as being processed;
             processNode(node->input2);
          }
       }
    }

    process node; //calculate coefficients, result ...
    mark node as processed;
}
```

Listing 3: Pseudocode: Recursion network Processing

Memory usage is hard to predict, but we expect that it will be roughly the same as the memory usage of the normal Recursive calculation order, because memory usage of any recursive calculation order depends mainly on the number and position of the nodes with **Number of Users** higher than one.

3.6.4 Parallelization of Discrete Differential Evolution

Parallelization of Differential Evolution is well known, evaluation of the individuals is processed by more threads.

The only real question is, what individuals and in what order should be processed by the threads.

From the memory usage point of view, it is not wise to process individuals with big Network Models at the same time. Because population contains individuals with Network Models of various size, a strategy in which only one thread works on a individual with big Network Model and the rest of the threads works on the individuals with smaller Network Models seems as reasonable.

The ordering rules are:

- Individuals in the 1st population will be ordered based on their length, because we do not have any other means of network size estimation.
- Individuals in every other population will be ordered based on their number of nodes in the previous generation, as this value obviously does not correspond to the real number of nodes in the new individual, but is the best approximation we have.

These rules use only approximate number of nodes, because the real number would require Network Model creation, which we want to be part of the parallelization. And approximate numbers are sufficient enough, because Network Models with different number of layer usually contain quite different number of nodes.

Time efficiency of the ordering algorithm has almost zero impact on the calculation time, mainly because the size of the population is usually relatively small and after the first ordering, order of the individuals will not change drastically.

Threads will iterate through the population and evaluate individuals on first-come, first-served basis, but one of them will iterate backwards. This will assure that this thread will process individuals from the most to the least complex ones, while the rest of the threads will evaluate individuals from the least to the most complex. Threads will stop then there are no more individuals to be evaluated.

Memory usage of this algorithm is dependent on the population and can not be evaluated by properties as maximal number of results in the memory.

3.7 Hybrid DDE Parallelization

This parallelization uses DDE parallelization on the individuals with reasonable complexity of the Network Models and parallelization of the Network Model creation and evaluation of the network on very complex individuals.

Implementation is relatively straight forward; use parallelization of DDE on individuals that have lower complexity, iterate to a predefined threshold, use parallelization of the Network Model creation and evaluation of the network on very complex individuals on the remaining unprocessed individuals.

4 Experimentation

This sections contains various experiments, that were carried out to evaluate performance of previously described operations.

- Any new abbreviations, that will appear in this section, will be used only in their respective sections, primarily in the columns names of tables.
- Measured time contains only time the described operation took, and it was measured with `clock_gettime` function, which gives time in nanosecond resolution. Additions of the time were also performed in precise format.
- As mentioned in the section 2, for GCC our application has two thread implementations, Open MP and our wrapper for platform specific threads, because all the test were carried out on machine with GNU/Linux, POSIX threads were used. GCC also uses POSIX threads when implementing Open MP, therefore when we will compare results of our thread implementation and Open MP threads, we are in fact comparing two different wrappers of the same POSIX threads.

4.1 Network Structure Optimization

In this section, we will show the results of the experiments that were created, to evaluate efficiency and parallelization suitability of the Network Structure Optimization.

Both experiments were performed on the same population, Table 1 contains the setting of the DE-GMDH.

Table 1: Network Structure Optimization, Parameters

Name of the Setting	**Value**
Input Data Columns	12
Training Rows	153
Testing Rows	173
Population Size	90
Number of Generations	50
Evaluations of Network Models with the same estimated number of layers	250

4.1.1 Efficiency Experiment

This experiment shows the sums of all non-redundant and redundant nodes that were in the Network Models with the same estimated number of layers (length of an individual). It also shows how much time it takes to create the Network Model and evaluate its fitness using Recursive Calculation Order, in-order to show the time difference between the Network Model creation and Evaluation. Table 2 contains the results of this experiment.

Table 2: Network Optimization, Efficiency Experiment 1

Estimated Layers	Sums of Nodes	Sums of Non-redundant Nodes	Network Model Creation Time [s]	Network Model Evaluation Time [s]
2	710	706	0.00564773	0.055459
3	1654	1645	0.00228569	0.148753
4	3614	3576	0.00276294	0.320835
5	7409	7009	0.00399294	0.61568
6	15018	13801	0.00557357	1.06765
7	30133	25274	0.00899713	1.8371
8	60817	45113	0.0157533	3.06036
9	122152	78667	0.0256925	5.30277
10	243214	141056	0.046876	9.52886
11	488583	259949	0.0918916	17.8091
12	979014	485005	0.185668	33.3113
13	1954144	889502	0.313751	61.1067
14	3912135	1579642	0.612596	109.037
15	7825179	2743585	1.36102	190.341
16	15640872	4846572	3.55693	336.159
17	31304341	8948418	8.02728	620.049
18	62577913	17114580	19.4081	1193.06
19	125153459	33397175	43.1933	2319.41

4.1.2 Parallelization Experiment

This experiments shows the execution times that the Network Structure Optimization took with different number of threads. Tables 3 and 4 show all times in seconds, tables 5 and 6 show single thread time in seconds and multithreaded times in percents of the single threaded time. Tables 3 and 4 use Open MP and tables 5 and 6 use our thread wrapper. Table 7 shows time differences between Open MP and our thread in percents, time of our thread wrapper version was used as a reference, therefore, if the number is positive, Open MP is better, than our thread wrapper. In case of the negative number, our thread wrapper is better, than Open MP

4.2 Calculation Orders: Recursive vs Layer-by-Layer

This experiment compares Recursive and Layer-by-Layer calculation orders. Maximal number of results in memory, calculation time of normal and parallelized versions will be compared.

If not stated otherwise, experiments were performed on the same population. Table 8 contains the setting of the DE-GMDH algorithm.

Table 3: Network Optimization, Parallelization Experiment Open MP Time[s]

Est. Layers	**Time 1 Thread[s]**	**Time 2 Threads[s]**	**Time 4 Threads[s]**	**Time 6 Threads[s]**	**Time 8 Threads[s]**
2	0.00564773	0.00691345	0.00108211	0.000972898	0.00112654
3	0.00228569	0.00380326	0.0016866	0.00163834	0.00164072
4	0.00276294	0.00455379	0.00253388	0.0027323	0.00244122
5	0.00399294	0.00587977	0.00379447	0.00368038	0.00363223
6	0.00557357	0.0105234	0.0058384	0.00561566	0.00550478
7	0.00899713	0.0125493	0.00982546	0.00928574	0.00909992
8	0.0157533	0.0716702	0.0240048	0.0227154	0.0218558
9	0.0256925	0.042622	0.123986	0.124659	0.110976
10	0.046876	0.0880508	0.251579	0.394898	0.523307
11	0.0918916	0.13078	0.404986	0.650847	0.958866
12	0.185668	0.210476	0.708756	1.10169	1.48818
13	0.313751	0.32473	1.038	1.55134	2.15686
14	0.612596	0.622222	1.32282	2.05681	2.55152
15	1.36102	1.09763	2.11408	2.76192	3.58418
16	3.55693	2.50654	3.32302	3.95656	4.86566
17	8.02728	5.40084	7.0505	7.69945	8.48063
18	19.4081	10.6937	9.40814	9.7441	11.6739
19	43.1933	26.0378	19.4897	17.5226	24.9218

Table 4: Network Optimization, Parallelization Experiment Our Thread Wrapper Time[s]

Est. Layers	**Time 1 Thread[s]**	**Time 2 Threads[s]**	**Time 4 Threads[s]**	**Time 6 Threads[s]**	**Time 8 Threads[s]**
2	0.00564773	0.00464443	0.00119946	0.00125216	0.00162827
3	0.00228569	0.00237392	0.00109716	0.00108986	0.00120303
4	0.00276294	0.00259842	0.00147404	0.00145802	0.00156475
5	0.00399294	0.00357253	0.00226347	0.00216174	0.00217669
6	0.00557357	0.00517256	0.00369664	0.00339692	0.00335306
7	0.00899713	0.00853227	0.00650458	0.00578312	0.00556928
8	0.0157533	0.0928306	0.0344918	0.0319672	0.0731931
9	0.0256925	0.194991	0.100906	0.0968116	0.179605
10	0.046876	0.121582	0.165276	0.196753	0.235016
11	0.0918916	0.137581	0.323492	0.214895	1.23816
12	0.185668	0.298786	0.335048	0.329237	0.524755
13	0.313751	0.514252	0.626358	0.650162	0.533249
14	0.612596	0.812328	0.84484	1.01663	0.97733
15	1.36102	1.43303	1.43774	1.56111	1.54492
16	3.55693	3.412	2.84577	2.96974	2.63489
17	8.02728	7.5542	6.5837	5.55917	4.99779
18	19.4081	13.4704	11.6072	11.7	9.65727
19	43.1933	24.8975	19.4432	19.7976	19.3642

Table 5: Network Optimization, Parallelization Experiment Open MP Time[%]

Est. Layers	**Time 1 Thread[s]**	**Time 2 Threads[%]**	**Time 4 Threads[%]**	**Time 6 Threads[%]**	**Time 8 Threads[%]**
2	0.00564773	122.411	19.160	17.226	19.947
3	0.00228569	166.394	73.790	71.678	71.782
4	0.00276294	164.817	91.710	98.891	88.356
5	0.00399294	147.254	95.029	92.172	90.966
6	0.00557357	188.809	104.752	100.755	98.766
7	0.00899713	139.481	109.207	103.208	101.142
8	0.0157533	454.954	152.380	144.195	138.738
9	0.0256925	165.893	482.577	485.196	431.939
10	0.046876	187.838	536.690	842.431	1116.364
11	0.0918916	142.320	440.721	708.277	1043.475
12	0.185668	113.361	381.733	593.366	801.527
13	0.313751	103.499	330.836	494.449	687.443
14	0.612596	101.571	215.937	335.753	416.509
15	1.36102	80.648	155.331	202.930	263.345
16	3.55693	70.469	93.424	111.235	136.794
17	8.02728	67.281	87.832	95.916	105.648
18	19.4081	55.099	48.475	50.206	60.150
19	43.1933	60.282	45.122	40.568	57.698

Table 6: Network Optimization, Parallelization Experiment Our Thread Wrapper Time[%]

Est. Layers	**Time 1 Thread[s]**	**Time 2 Threads[%]**	**Time 4 Threads[%]**	**Time 6 Threads[%]**	**Time 8 Threads[%]**
2	0.00564773	82.235	21.238	22.171	28.831
3	0.00228569	103.860	48.001	47.682	52.633
4	0.00276294	94.045	53.350	52.771	56.634
5	0.00399294	89.471	56.687	54.139	54.513
6	0.00557357	92.805	66.324	60.947	60.160
7	0.00899713	94.833	72.296	64.277	61.901
8	0.0157533	589.277	218.950	202.924	464.621
9	0.0256925	758.941	392.745	376.809	699.056
10	0.046876	259.369	352.581	419.731	501.357
11	0.0918916	149.721	352.037	233.857	1347.414
12	0.185668	160.925	180.455	177.326	282.631
13	0.313751	163.904	199.635	207.222	169.959
14	0.612596	132.604	137.911	165.954	159.539
15	1.36102	105.291	105.637	114.701	113.512
16	3.55693	95.925	80.006	83.492	74.078
17	8.02728	94.107	82.017	69.253	62.260
18	19.4081	69.406	59.806	60.284	49.759
19	43.1933	57.642	45.014	45.835	44.831

Table 7: Network Optimization, Parallelization Experiment Open MP vs Our Thread Wrapper

Est. Layers	Time 2 Threads[%]	Time 4 Threads[%]	Time 6 Threads[%]	Time 8 Threads[%]
2	-48.85	9.78	22.30	30.81
3	-60.21	-53.72	-50.33	-36.38
4	-75.25	-71.90	-87.40	-56.01
5	-64.58	-67.64	-70.25	-66.87
6	-103.45	-57.94	-65.32	-64.17
7	-47.08	-51.05	-60.57	-63.39
8	22.79	30.40	28.94	70.14
9	78.14	-22.87	-28.76	38.21
10	27.58	-52.22	-100.71	-122.67
11	4.94	-25.19	-202.87	22.56
12	29.56	-111.54	-234.62	-183.60
13	36.85	-65.72	-138.61	-304.48
14	23.40	-56.58	-102.32	-161.07
15	23.40	-47.04	-76.92	-132.00
16	26.54	-16.77	-33.23	-84.66
17	28.51	-7.09	-38.50	-69.69
18	20.61	18.95	16.72	-20.88
19	-4.58	-0.24	11.49	-28.70

Table 8: Calculation Orders: Recursive vs Layer-by-Layer, Parameters

Name of the Setting	Value
Input Data Columns	12
Training Rows	153
Testing Rows	173
Population Size	90
Number of Generations	50
Evaluations of Network Models with the same estimated number of layers	250

4.2.1 Normal Versions

Table 9 shows averages of maximal number of results in memory and sums of calculation time of the the Network Models with the same estimated number of layers (length of an individual) for Layer-by-Layer and Recursive calculation orders.

4.2.2 Parallelized Versions

The following results were obtained with Open MP implementation: Tables 10, 11, 12, 13 shows the same attributes, but with use of 2, 4, 6 and 8 threads respectively and Tables 14 and 15 shows single thread time in seconds and multithreaded times in percents of the single threaded time, for Layer-by-Layer and Recursive calculation orders, respectively.

Table 9: Calculation Orders: Recursive vs Layer-by-Layer

	Layer-by-Layer		Recursive	
Estimated Layers	**AVG Max Results in Memory**	**Total Calculation Time [s]**	**AVG Max Results in Memory**	**Total Calculation Time [s]**
2	1.8	0.0635467	1.8	0.0533856
3	3.408	0.142611	2.592	0.143329
4	6.816	0.307802	3.852	0.310276
5	13.264	0.613552	5.044	0.60832
6	23.616	1.03994	7.708	1.02648
7	41.256	1.80649	13.884	1.7874
8	69.408	3.05694	29.98	3.0212
9	129.824	5.25472	54.368	5.10315
10	246.788	9.22221	75.944	9.07882
11	468.928	16.8824	96.496	16.7701
12	861.556	31.4117	153.244	31.2897
13	1464.18	57.5351	330.708	57.3733
14	2295.86	102.085	813.304	101.879
15	4206.86	177.298	1674.64	176.91
16	8245.35	313.168	2569.78	312.199
17	16380.7	579.347	2999.96	575.88
18	32652.2	1115.42	3144.77	1100.72
19	65013.5	2176.36	3352.89	2147.49

The following results were obtained with the thread wrapper: Tables 16, 17, 18, 19 shows the same attributes, but with use of 2, 4, 6 and 8 threads respectively and Tables 20 and 21 shows single thread time in seconds and Open MP multithreaded times in percents of the single threaded time, for Layer-by-Layer and Recursive calculation orders, respectively.

Tables 22 and 23 shows time differences between Open MP and the thread wrapper in percents, for Layer-by-Layer and Recursive orders, respectively. Calculation time of our thread wrapper version was used as a reference, therefore, if the number is positive then Open MP is better than the thread wrapper. In case of the negative number, the thread wrapper is better than Open MP

Parallelized Recursive calculation order has a slightly different implementation of thread scheduling for Open MP and the thread wrapper, because Open MP implementation uses only `#pragma omp atomic` for atomic memory operations. The thread wrapper uses GCC `__atomic` built-in functions, which have more operations than the `#pragma omp atomic`. Because of this, a third parallel implementation of the Recursive calculation order was implemented. This only uses Open MP threads and GCC `__atomic` built-in functions, which makes it GCC compiler dependent.

Tables 25 and 26 show time differences between GCC `__atomic` and Open MP implementations in percentage. GCC `__atomic` implementations was used as a reference, therefore, if

the number is positive, Open MP is better than GCC `__atomic`. In case of the negative number, GCC `__atomic` is better than Open MP. Result in these tables were obtained on different populations, however the setting of the DE-GMDH was the same as the one in Table 24.

Table 10: Calculation Orders: Recursive vs Layer-by-Layer, 2 Threads, Open MP

	Layer-by-Layer		**Recursive**	
Estimated Layers	**AVG Max Results in Memory**	**Total Calculation Time [s]**	**AVG Max Results in Memory**	**Total Calculation Time [s]**
2	1.8	0.0413943	1.8	0.0644155
3	3.448	0.0774617	2.608	0.10992
4	6.856	0.142879	4.656	0.193135
5	13.3	0.270528	6.692	0.339617
6	23.9	0.494019	9.508	0.593614
7	41.444	0.892207	16.116	1.0274
8	69.932	1.54068	32.416	1.71263
9	131.46	2.62388	57.556	2.82316
10	247.832	4.62456	80.2	4.81893
11	469.736	8.51364	101.16	8.78873
12	861.86	15.9214	158.172	16.1496
13	1464.84	29.3472	336.088	29.539
14	2327.08	52.0548	819.7	52.5321
15	4254.62	90.4988	1680.97	91.6234
16	8269.09	160.435	2577.13	160.679
17	16383.5	297.582	3008.41	297.75
18	32653.2	571.56	3154.21	567.252
19	65013.3	1118.48	3362.62	1103.15

4.3 DDE vs Parallelized DDE vs Parallelized Recursive

In this section, we will compare the peak memory usage and time of the:

- Single threaded DDE with single threaded Recursive calculation order for fitness evaluation (SDSR)
- Multi-threaded DDE with single threaded Recursive calculation order for fitness evaluation (MDSR)
- Single threaded DDE with multi-threaded Recursive calculation order for fitness evaluation (SDMR)

The experiments had the following characteristics:

- Open MP was used for the threads, GCC Open MP version of the Recursive calculation order was used.

Table 11: Calculation Orders: Recursive vs Layer-by-Layer, 4 Threads, Open MP

	Layer-by-Layer		Recursive	
Estimated Layers	**AVG Max Results in Memory**	**Total Calculation Time [s]**	**AVG Max Results in Memory**	**Total Calculation Time [s]**
2	1.8	0.0436889	1.8	0.0644973
3	3.448	0.0598583	3.328	0.0641859
4	6.924	0.0951029	6.104	0.109727
5	13.404	0.161848	9.408	0.183499
6	24.208	0.273734	13.6	0.305936
7	42.268	0.477813	21.212	0.52221
8	71.668	0.803832	37.572	0.876377
9	132.236	1.35275	64.82	1.455
10	249.092	2.36043	87.932	2.51472
11	471.052	4.3028	110.912	4.55448
12	863.18	8.02227	168.584	8.45031
13	1465.73	14.6815	349.404	15.1784
14	2367.49	26.0956	826.148	26.6985
15	4280.29	45.263	1691.08	46.1575
16	8279.22	80.3646	2591.98	81.3006
18	32648.6	288.455	3169.9	288.365
19	65009.4	564.903	3381.39	562.95

- Two tests were run, each of them is composed of 50 complete runs of the DE-GMDH algorithms, where each DE-GMDH algorithm was evaluated by the three aforementioned DE-GMDH implementations, but only single threaded DDE with single threaded Recursive calculation order for fitness evaluation actually performed DDE mutation, crossover and selection on the population and saved all the populations that were created. The other two implementations used these population instead of evolving their own in order to ensure unbiased experiments and conformity.
- Peak memory usage was measured by `time` utility.

4.3.1 First Experiment

This experiment contains individuals with a relatively low number of network layers, as such individuals are used for the processing of the data, that has relatively low number of input columns. Peak memory usage and time will therefore be relatively low. Table 27 contains the setting of the DE-GMDH and Table 28 contains the results.

Table 12: Calculation Orders: Recursive vs Layer-by-Layer, 6 Threads, Open MP

	Layer-by-Layer		**Recursive**	
Estimated Layers	**AVG Max Results in Memory**	**Total Calculation Time [s]**	**AVG Max Results in Memory**	**Total Calculation Time [s]**
2	1.8	0.0394908	1.8	0.0465021
3	3.448	0.0585158	3.176	0.0686553
4	6.924	0.0884087	5.968	0.104384
5	13.444	0.143729	9.816	0.161255
6	24.4	0.221876	15.004	0.265233
7	42.9	0.36631	23.404	0.444137
8	72.724	0.595734	39.9	0.670227
9	133.404	0.961534	67.036	1.09244
10	250.204	1.64321	92.192	1.82829
11	472.256	2.95933	116.9	3.23864
12	864.232	5.45814	175.972	5.84641
13	1466.1	9.86881	353.328	10.4526
14	2366.21	17.4328	833.592	18.2647
15	4284.2	30.2845	1698.94	31.2577
16	8282.35	54.0738	2598.34	54.9852
17	16388.8	101.091	3036.5	101.906
18	32648.4	193.643	3184.55	194.395
19	65010.9	379.55	3394.59	379.404

Table 13: Calculation Orders: Recursive vs Layer-by-Layer, 8 Threads, Open MP

	Layer-by-Layer		**Recursive**	
Estimated Layers	**AVG Max Results in Memory**	**Total Calculation Time [s]**	**AVG Max Results in Memory**	**Total Calculation Time [s]**
2	1.8	0.131515	1.8	0.130929
3	3.448	0.139401	3.32	0.148327
4	6.924	0.149572	6.508	0.192693
5	13.512	0.219489	11.312	0.263665
6	24.736	0.320469	17.296	0.350298
7	43.468	0.480617	26.152	0.521325
8	72.832	0.748184	43.608	0.775674
9	131.808	1.2313	70.712	1.16135
10	248.772	2.07064	97.916	1.84837
11	470.908	3.64385	121.996	3.14748
12	863.088	6.61568	182.632	5.50735
13	1466.73	11.7858	362.204	9.6977
14	2369.79	20.0849	838.776	16.8259
15	4298.19	34.0889	1706.34	28.408
16	8280.08	59.4129	2607.29	48.5687
17	16381.5	106.111	3046.37	86.7313
18	32644.7	196.862	3197.31	157.847
19	65008.7	362.23	3408.77	300.778

Table 14: Calculation Orders: Layer-by-Layer, Open MP Parallelization Time Comparison

Estimated Layers	**Time 1 Thread [s]**	**Time 2 Threads [%]**	**Time 4 Threads [%]**	**Time 6 Threads [%]**	**Time 8 Threads [%]**
2	0.0635467	65.140	68.751	62.145	206.958
3	0.142611	54.317	41.973	41.032	97.749
4	0.307802	46.419	30.897	28.723	48.594
5	0.613552	44.092	26.379	23.426	35.773
6	1.03994	47.505	26.322	21.335	30.816
7	1.80649	49.389	26.450	20.277	26.605
8	3.05694	50.399	26.295	19.488	24.475
9	5.25472	49.934	25.744	18.298	23.432
10	9.22221	50.146	25.595	17.818	22.453
11	16.8824	50.429	25.487	17.529	21.584
12	31.4117	50.686	25.539	17.376	21.061
13	57.5351	51.007	25.517	17.153	20.485
14	102.085	50.992	25.563	17.077	19.675
15	177.298	51.043	25.529	17.081	19.227
16	313.168	51.230	25.662	17.267	18.972
17	579.347	51.365	26.003	17.449	18.316
18	1115.42	51.242	25.861	17.361	17.649
19	2176.36	51.392	25.956	17.440	16.644

Table 15: Calculation Orders: Recursive, Open MP Parallelization Time Comparison

Estimated Layers	**Time 1 Thread [s]**	**Time 2 Threads [%]**	**Time 4 Threads [%]**	**Time 6 Threads [%]**	**Time 8 Threads [%]**
2	0.0533856	120.661	120.814	87.106	245.252
3	0.143329	76.691	44.782	47.900	103.487
4	0.310276	62.246	35.364	33.642	62.104
5	0.60832	55.829	30.165	26.508	43.343
6	1.02648	57.830	29.804	25.839	34.126
7	1.7874	57.480	29.216	24.848	29.167
8	3.0212	56.687	29.008	22.184	25.674
9	5.10315	55.322	28.512	21.407	22.758
10	9.07882	53.079	27.699	20.138	20.359
11	16.7701	52.407	27.158	19.312	18.768
12	31.2897	51.613	27.007	18.685	17.601
13	57.3733	51.486	26.456	18.219	16.903
14	101.879	51.563	26.206	17.928	16.516
15	176.91	51.791	26.091	17.669	16.058
16	312.199	51.467	26.041	17.612	15.557
17	575.88	51.703	26.154	17.696	15.061
18	1100.72	51.535	26.198	17.661	14.340
19	2147.49	51.369	26.214	17.667	14.006

Table 16: Calculation Orders: Recursive vs Layer-by-Layer, 2 Threads, Our Thread Wrapper

	Layer-by-Layer		**Recursive**	
Estimated Layers	**AVG Max Results in Memory**	**Total Calculation Time [s]**	**AVG Max Results in Memory**	**Total Calculation Time [s]**
2	1.8	0.0620068	1.8	0.0616032
3	3.408	0.141706	3.3	0.10642
4	6.816	0.257951	5.284	0.199333
5	13.3	0.296955	7.184	0.289725
6	23.9	0.513219	9.924	0.534232
7	41.452	0.910064	17.016	0.967481
8	69.96	1.55365	33.512	1.62957
9	131.46	2.65444	58.74	2.70663
10	247.812	4.66013	81.056	4.71135
11	469.752	8.62079	102.22	8.61189
12	861.984	15.9195	163.044	15.9546
13	1464.91	29.124	339.952	29.1911
14	2327.32	51.8865	810.608	51.8219
15	4256.1	89.9059	1680.54	90.2322
16	8272.41	159.451	2580.08	160.144
17	16382.9	294.598	3011.41	296.55
18	32649.0	566.412	3153.76	567.528
19	65009.1	1105.79	3366.02	1097.77

Table 17: Calculation Orders: Recursive vs Layer-by-Layer, 4 Threads, Our Thread Wrapper

	Layer-by-Layer		**Recursive**	
Estimated Layers	**AVG Max Results in Memory**	**Total Calculation Time [s]**	**AVG Max Results in Memory**	**Total Calculation Time [s]**
2	1.8	0.0620958	1.8	0.698503
3	3.408	0.141628	2.592	0.177115
4	6.816	0.187124	4.592	0.183702
5	13.304	0.31507	7.88	0.29297
6	24.192	0.362498	11.324	0.465802
7	42.32	0.569065	18.004	0.740337
8	71.496	0.906163	35.072	1.15605
9	132.056	1.47505	61.532	1.79886
10	248.484	2.88253	85.6	2.92538
11	470.624	5.0688	107.824	5.64077
12	862.876	9.06685	165.948	9.74969
13	1465.8	15.5796	343.772	16.9924
14	2367.61	27.2492	826.8	28.614
15	4280.26	47.5063	1690.24	48.7336
16	8277.45	83.316	2588.91	84.1058
17	16386.9	151.046	3022.6	154.358
18	32653.4	289.224	3170.05	292.472
19	65011.5	565.998	3378.96	566.377

Table 18: Calculation Orders: Recursive vs Layer-by-Layer, 6 Threads, Our Thread Wrapper

	Layer-by-Layer		**Recursive**	
Estimated Layers	**AVG Max Results in Memory**	**Total Calculation Time [s]**	**AVG Max Results in Memory**	**Total Calculation Time [s]**
2	1.8	0.0619678	1.8	2.04451
3	3.408	0.13736	2.592	0.302095
4	6.816	0.187115	4.58	0.301572
5	13.3	0.306775	8.156	0.351364
6	24.18	0.420339	12.624	0.514816
7	42.724	0.529627	19.5	0.791623
8	72.408	0.788135	36.344	1.20048
9	132.876	1.21439	63.372	1.93289
10	249.004	2.29329	87.608	3.0811
11	470.908	4.05044	112.044	5.70715
12	863.364	7.20056	169.704	8.78598
13	1465.84	12.9971	347.208	14.2531
14	2368.03	21.3116	828.588	22.526
15	4283.71	34.4251	1690.82	36.5143
16	8280.19	59.068	2590.3	61.6202
17	16387.3	106.173	3029.6	110.298
18	32652.4	197.557	3183.63	204.329
19	65013.3	386.445	3393.17	391.226

Table 19: Calculation Orders: Recursive vs Layer-by-Layer, 8 Threads, Our Thread Wrapper

	Layer-by-Layer		**Recursive**	
Estimated Layers	**AVG Max Results in Memory**	**Total Calculation Time [s]**	**AVG Max Results in Memory**	**Total Calculation Time [s]**
2	1.8	0.062171	1.8	3.29968
3	3.408	0.112236	2.592	0.745072
4	6.816	0.18639	4.58	0.557783
5	13.3	0.314561	8.144	0.566513
6	24.184	0.436929	13.848	0.643753
7	43.372	0.635244	21.196	0.949483
8	72.844	0.875207	38.472	1.28886
9	132.348	1.33357	65.7	2.09263
10	249.088	2.53767	91.316	3.1434
11	470.716	4.56694	116.52	6.12888
12	863.012	7.53431	175.128	9.01088
13	1466.6	12.9126	352.04	13.423
14	2378.2	21.7902	833.656	21.305
15	4301.45	35.9617	1697.03	33.5722
16	8277.63	60.8353	2597.31	55.834
17	16379.6	107.412	3036.73	96.1807
18	32644.6	196.197	3193.99	169.676
19	65008.9	363.6	3406.16	310.509

Table 20: Calculation Orders: Layer-by-Layer, Our Thread Wrapper Parallelization Time Comparison

Estimated Layers	**Time 1 Thread [s]**	**Time 2 Threads [%]**	**Time 4 Threads [%]**	**Time 6 Threads [%]**	**Time 8 Threads [%]**
2	0.06	97.58	97.72	97.52	97.84
3	0.14	99.37	99.31	96.32	78.70
4	0.31	83.80	60.79	60.79	60.56
5	0.61	48.40	51.35	50.00	51.27
6	1.04	49.35	34.86	40.42	42.01
7	1.81	50.38	31.50	29.32	35.16
8	3.06	50.82	29.64	25.78	28.63
9	5.25	50.52	28.07	23.11	25.38
10	9.22	50.53	31.26	24.87	27.52
11	16.88	51.06	30.02	23.99	27.05
12	31.41	50.68	28.86	22.92	23.99
13	57.54	50.62	27.08	22.59	22.44
14	102.09	50.83	26.69	20.88	21.35
15	177.30	50.71	26.79	19.42	20.28
16	313.17	50.92	26.60	18.86	19.43
17	579.35	50.85	26.07	18.33	18.54
18	1115.42	50.78	25.93	17.71	17.59
19	2176.36	50.81	26.01	17.76	16.71

Table 21: Calculation Orders: Recursive, Our Thread Wrapper Parallelization Time Comparison

Estimated Layers	**Time 1 Thread [s]**	**Time 2 Threads [%]**	**Time 4 Threads [%]**	**Time 6 Threads [%]**	**Time 8 Threads [%]**
2	0.0533856	115.39	1308.41	3829.70	6180.84
3	0.143329	74.62	123.57	210.77	519.83
4	0.310276	64.76	59.21	97.19	179.77
5	0.60832	47.22	48.16	57.76	93.13
6	1.02648	51.37	45.38	50.15	62.71
7	1.7874	53.56	41.42	44.29	53.12
8	3.0212	53.31	38.26	39.74	42.66
9	5.10315	51.51	35.25	37.88	41.01
10	9.07882	51.09	32.22	33.94	34.62
11	16.7701	51.01	33.64	34.03	36.55
12	31.2897	50.79	31.16	28.08	28.80
13	57.3733	50.74	29.62	24.84	23.40
14	101.879	50.76	28.09	22.11	20.91
15	176.91	50.89	27.55	20.64	18.98
16	312.199	51.14	26.94	19.74	17.88
17	575.88	51.19	26.80	19.15	16.70
18	1100.72	50.88	26.57	18.56	15.42
19	2147.49	50.44	26.37	18.22	14.46

Table 22: Calculation Orders: Layer-by-Layer, Our Thread Wrapper vs Open MP

Estimated Layers	**Time 2 Threads [%]**	**Time 4 Threads [%]**	**Time 6 Threads [%]**	**Time 8 Threads [%]**
2	33.24	29.64	36.27	-111.54
3	45.34	57.74	57.40	-24.20
4	44.61	49.18	52.75	19.75
5	8.90	48.63	53.15	30.22
6	3.74	24.49	47.21	26.65
7	1.96	16.04	30.84	24.34
8	0.83	11.29	24.41	14.51
9	1.15	8.29	20.82	7.67
10	0.76	18.11	28.35	18.40
11	1.24	15.11	26.94	20.21
12	-0.01	11.52	24.20	12.19
13	-0.77	5.76	24.07	8.73
14	-0.32	4.23	18.20	7.83
15	-0.66	4.72	12.03	5.21
16	-0.62	3.54	8.46	2.34
17	-1.01	0.26	4.79	1.21
18	-0.91	0.27	1.98	-0.34
19	-1.15	0.19	1.78	0.38

Table 23: Calculation Orders: Recursive, Our Thread Wrapper vs Open MP

Estimated Layers	**Time 2 Threads** [%]	**Time 4 Threads** [%]	**Time 6 Threads** [%]	**Time 8 Threads** [%]
2	-4.57	90.77	97.73	96.03
3	-3.29	63.76	77.27	80.09
4	3.11	40.27	65.39	65.45
5	-17.22	37.37	54.11	53.46
6	-11.12	34.32	48.48	45.59
7	-6.19	29.46	43.90	45.09
8	-5.10	24.19	44.17	39.82
9	-4.31	19.12	43.48	44.50
10	-2.28	14.04	40.66	41.20
11	-2.05	19.26	43.25	48.65
12	-1.22	13.33	33.46	38.88
13	-1.19	10.68	26.66	27.75
14	-1.37	6.69	18.92	21.02
15	-1.54	5.29	14.40	15.38
16	-0.33	3.34	10.77	13.01
17	-0.40	2.42	7.61	9.82
18	0.05	1.40	4.86	6.97
19	-0.49	0.61	3.02	3.13

Table 24: Calculation Orders: Recursive Open MP vs GCC atomic Open MP, Parameters

Name of the Setting	**Value**
Input Data Columns	12
Training Rows	153
Testing Rows	173
Population Size	90
Number of Generations	40
Evaluations of Network Models with the same estimated number of layers	200

Table 25: Calculation Orders: Recursive Open MP vs GCC atomic Open MP, Results 1

Estimated Layers	**Difference 2 Threads [%]**	**Difference 4 Threads [%]**	**Difference 6 Threads [%]**	**Difference 8 Threads [%]**
2	-50.52	-168.08	-49.68	1.09
3	-10.03	9.82	-12.88	6.94
4	-10.13	-13.88	-11.64	-5.28
5	-19.97	-5.24	-7.13	-2.96
6	-14.73	-7.83	-8.14	-1.11
7	-11.59	-6.84	-5.98	-4.88
8	-8.70	-7.12	-7.50	-11.61
9	-5.81	-4.54	-3.32	-7.05
10	-4.41	-2.70	-2.43	-1.14
11	-1.21	-0.56	-0.70	-4.92
12	-1.24	-0.88	-2.46	0.33
13	0.36	-0.76	-1.85	-2.93
14	1.15	-0.72	-1.29	-3.73
15	-0.27	-0.68	-1.28	-1.71
16	0.16	-0.41	-1.01	-1.83
17	1.12	0.53	-0.15	-1.40
18	0.96	-0.08	-0.18	-2.45
19	0.02	0.05	-0.63	-2.37

Table 26: Calculation Orders: Recursive Open MP vs GCC atomic Open MP, Results 2

Estimated Layers	**Difference 2 Threads [%]**	**Difference 4 Threads [%]**	**Difference 6 Threads [%]**	**Difference 8 Threads [%]**
2	9.41	57.78	17.49	26.46
3	-2.43	-0.33	3.09	15.31
4	-0.80	-7.90	3.65	3.87
5	15.64	-2.34	-0.17	1.50
6	13.21	-2.70	-0.53	3.42
7	9.95	-4.66	-3.55	5.29
8	6.11	-4.16	-1.02	-2.72
9	5.23	-1.52	-2.20	-1.16
10	3.34	-0.83	-1.40	-2.01
11	1.79	-3.10	-2.66	1.60
12	-0.11	-2.21	-0.41	-5.45
13	0.43	0.99	-0.03	-2.19
14	-0.07	-0.27	-0.18	-2.06
15	0.02	-0.85	-0.88	-2.65
16	-0.91	-1.11	-1.19	-1.51
17	-0.62	-1.11	-0.93	-1.72
18	0.18	-0.40	-0.16	-1.95
19	-0.04	-0.37	-0.50	-1.48

Table 27: DDE vs Parallelized DDE vs Parallelized Recursive, Experiment 1, Parameters

Name of the Setting	**Value**
Input Data Columns	12
Training Rows	153
Testing Rows	173
Minimal Network Layers	2
Maximal Network Layers	10
Population Size	100
Number of Generations	50
Number of Threads	2

Table 28: DDE vs Parallelized DDE vs Parallelized Recursive, Experiment 1, Results

Run No.	**SDSR Time [s]**	**MDSR Time [s]**	**SDMR Time [s]**	**SDSR Memory [kB]**	**MDSR Memory [kB]**	**SDMR Memory [kB]**
1	72.144971	36.429784	37.196912	2432	2216	2288
2	75.319033	37.800371	38.712666	2432	2220	2276
3	77.916903	38.728197	40.175938	2464	2216	2288
4	75.845685	38.107765	39.277367	2460	2216	2292
5	74.788964	37.716491	38.727239	2436	2212	2288
6	89.842349	45.982315	46.327983	2552	2244	2320
7	74.447769	37.180409	38.177029	2432	2248	2292
8	68.787646	34.303361	35.489714	2400	2220	2276
9	54.716051	27.035156	28.137861	2292	2272	2244
10	69.577027	34.342872	35.681414	2404	2212	2272
11	63.607296	32.912881	33.014962	2372	2244	2268
12	83.60376	41.52031	43.006371	2488	2216	2304
13	76.968924	37.961813	39.462171	2464	2216	2292
14	80.405331	39.938672	41.192675	2464	2216	2300
15	73.72025	36.912006	37.979295	2436	2212	2280
16	72.842629	36.657961	37.381123	2424	2344	2280
17	86.941341	43.220697	44.504508	2528	2248	2324
18	83.236413	42.235137	42.697887	2492	2244	2216
19	67.241456	33.237194	34.881004	2384	2216	2272
20	80.830018	39.821737	41.599928	2476	2220	2252
21	65.273251	32.361853	33.496273	2364	2216	2264
22	66.222598	33.033106	33.942148	2384	2216	2268
23	80.811646	39.897599	41.518044	2476	2216	2200
24	62.51169	31.177421	32.219241	2340	2216	2252
25	74.468791	37.515019	38.453183	2432	2212	2276
26	83.264358	41.658741	42.762068	2500	2220	2208
27	67.967277	33.555109	34.804927	2396	2216	2272
28	74.463432	36.85334	38.217153	2428	2216	2284
29	62.487342	31.45746	32.195139	2356	2216	2260
30	76.6586	38.40494	39.381539	2444	2212	2292
31	72.83361	36.435453	37.551097	2420	2208	2284

32	76.903995	38.39336	39.745739	2436	2216	2288
33	75.127509	37.431717	38.649438	2436	2212	2192
34	76.749725	39.087909	39.449867	2448	2248	2292
35	89.715672	44.823879	45.956413	2540	2236	2324
36	77.605885	39.615625	39.937452	2448	2212	2248
37	71.071347	35.142321	36.660403	2408	2212	2200
38	65.750773	32.83851	33.977802	2380	2220	2184
39	85.869309	42.806199	44.025901	2504	4264	2316
40	62.807749	32.054781	32.425964	2364	2244	2256
41	68.096225	33.650548	35.092288	2400	2212	2272
42	63.423721	32.038478	32.632758	2368	2212	2268
43	73.269747	36.433659	37.688148	2424	2212	2280
44	69.32489	34.630723	36.013388	2384	2216	2276
45	81.18059	40.254515	41.969677	2484	2352	2296
46	82.701647	42.11847	42.627861	2492	2244	2300
47	51.675271	25.661338	26.63041	2272	2216	2188
48	65.467562	33.033114	33.902445	2376	2212	2264
49	93.18408	46.613009	47.61991	2572	2252	2300
50	87.866061	43.433586	45.048104	2532	2228	2316

4.3.2 Second Experiment

This experiment contains individuals with a higher number of network layers, as such individuals are used for the processing of the data that has a high number of input columns. Peak memory usage and time will be noticeable. Table 29 contains the setting of the DE-GMDH and Table 30 contains the results.

Input data used in this test have a relatively low number of columns and small population and just one generation. These properties imply that this test does not make sense from the usability point of view, but is sufficient to demonstrate the fitness evaluation time and peak memory usage.

Table 29: DDE vs Parallelized DDE vs Parallelized Recursive, Experiment 2, Parameters

Name of the Setting	**Value**
Input Data Columns	12
Training Rows	153
Testing Rows	173
Minimal Network Layers	12
Maximal Network Layers	22
Population Size	50
Number of Generations	1
Number of Threads	4

Table 30: DDE vs Parallelized DDE vs Parallelized Recursive, Experiment 2, Results

Run No.	**SDSR Time [s]**	**MDSR Time [s]**	**SDMR Time [s]**	**SDSR Memory [kB]**	**MDSR Memory [kB]**	**SDMR Memory [kB]**
1	347.474101	101.452585	94.347821	112416	164416	113932
2	243.553559	67.601088	65.702866	85192	127736	88312
3	208.902436	56.64397	56.479517	76704	105364	77972
4	358.255344	100.523337	98.606927	113200	196968	116700
5	334.217903	101.382269	90.085571	107232	165112	110356
6	321.649231	96.26101	86.76462	104260	184024	107824
7	336.753791	94.813532	89.603713	107960	187752	111236
8	275.198062	78.925681	74.01078	92896	158948	96420
9	271.462147	84.255149	73.971467	91952	142072	95268
10	267.681437	69.01173	72.412532	90992	127444	94592
11	266.679764	78.309731	70.163186	90812	159680	94288
12	408.223619	113.980865	111.011151	125316	209064	128956
13	260.072526	72.401917	70.038873	89212	155300	92388
14	236.260033	69.979058	63.932383	91040	149468	94096
15	300.181988	80.412991	80.670056	98920	136164	102300
16	334.607046	96.428045	90.312194	107432	172372	111980
17	317.138547	101.256294	84.420527	102940	141748	106204
18	289.357318	74.742299	78.429516	96540	176256	100132
19	301.124623	90.320637	80.505048	99336	179152	102948
20	340.271151	101.499674	91.626937	108876	192488	112628
21	289.2631	87.071559	78.412686	96340	146316	99816
22	337.154456	95.083517	92.141551	108116	173040	111220
23	243.987809	68.656421	66.056924	85280	127936	87312
24	367.590587	103.91532	99.812202	115424	169364	118604
25	326.935536	100.981224	87.989117	105332	159348	108804
26	346.814753	98.732047	93.255529	110404	194280	114116
27	341.250759	95.214611	92.203722	109192	188964	112856
28	469.327802	134.301501	126.818919	140232	220056	143864
29	413.135435	119.682681	111.258299	128364	210148	130920
30	247.158568	68.008477	66.182392	85712	107308	87316
31	389.194237	108.261053	105.483307	122988	204588	124432

32	260.09436	72.241684	70.493575	91252	131848	93832
33	424.826082	111.651051	115.03759	129484	213184	132912
34	224.706459	67.852106	60.365501	80672	123072	84120
35	515.995185	138.08794	141.308515	151668	235376	155256
36	483.196842	136.805858	134.673193	150932	227172	154384
37	368.690695	101.459765	99.897661	115672	184464	119104
38	356.372132	101.863919	96.29449	112880	196560	116064
39	288.129246	92.044912	77.929599	96016	146204	99268
40	473.96483	128.231626	128.873735	143428	225152	145008
41	443.188531	115.453147	119.962774	134000	217764	137420
42	283.835074	84.089337	76.821113	95000	159964	98064
43	383.816724	107.587154	104.318812	126952	203092	130060
44	435.908229	122.74903	118.279415	139576	215708	142900
45	312.68656	96.219482	84.846822	101972	151996	105476
46	402.968683	116.574358	109.170688	123908	207516	127500
47	308.362236	95.946845	83.391406	108104	150736	111540
48	526.41706	136.514068	143.46278	161576	237676	164596
49	294.652754	87.298762	80.023181	97712	147768	100780
50	336.116891	101.228292	91.265115	109520	146364	111184

4.4 Verification of the DE-GMDH implementation

This experiment was performed to verify if the DE-GMDH implementation (with Hybrid DDE Parallelization) can repeatedly find GMDH network models, which reasonably describes the problem (their fitness is within feasible bounds). Finding the best possible network model for the given problem was not the aim of this experiment.

Three data sets were used during the test [13, 8, 11] respectively. The DE-GMDH evaluated each of them 100 times. Calculation time, memory usage, fitness of the final network model and maximal prediction error is given for each evaluation. Graphs with predicted and measured values will be shown for three randomly selected results for each dataset.

Fitness of the result fi is calculated by Equation 7

$$fi = \frac{r}{\sum_{i=1}^{r} (y - z)^2} \tag{7}$$

where r is number of rows in the input data, y is the measured value of the result and z is the predicted value of the result.

4.4.1 First Data Set

Table 31 shows setting for the DE-GMDH, that were used during this experiment of the data set [13]. Table 32 shows the results obtained from this experiment. Figures 8, 9, 10 show graphs with measured and predicted values of the runs 15, 18 and 99 respectively.

Table 31: DE-GMDH Test, Data Set [13], Parameters

Name of the Setting	**Value**
Input Data Columns	22
Training Rows	17
Testing Rows	15
Minimal Network Layers	3
Maximal Network Layers	5
Population Size	100
Number of Generations	100
Number of Threads	2

Table 32: DE-GMDH Test, Data Set[13], Results

Run No.	**Time [s]**	**Memory [kB]**	**Fitness**	**Max Error**
1	0.89	3252	366.536	0.142896
2	0.87	3188	421.025	0.136765
3	0.85	3280	395.560	0.104721
4	0.86	3180	364.866	0.142869
5	1.15	3352	302.812	0.154906
6	0.86	3280	416.431	0.118953
7	0.84	3200	383.136	0.162695
8	0.88	3156	354.240	0.14157
9	0.83	3180	472.850	0.124564
10	0.88	3156	558.526	0.150838
11	0.82	3276	348.532	0.133302
12	0.85	3204	328.848	0.133249
13	0.90	3264	339.227	0.109951
14	0.84	3360	422.763	0.108903
15	0.79	3232	331.022	0.128025
16	0.83	3276	368.401	0.121646
17	0.91	3268	379.751	0.119311
18	0.79	3180	369.177	0.115968
19	0.76	3200	317.858	0.110263
20	1.07	3180	363.693	0.116237
21	0.84	3320	345.002	0.126716
22	0.89	3200	370.019	0.133486
23	0.87	3156	331.782	0.160003
24	0.83	3228	400.554	0.140968
25	0.72	3260	345.306	0.148924
26	0.78	3316	338.737	0.139925
27	0.81	3180	362.480	0.111832
28	0.83	3272	334.108	0.168974
29	0.93	3264	399.558	0.126944
30	0.81	3276	315.472	0.133816
31	0.92	3396	353.222	0.144317
32	0.90	3284	367.325	0.140089
33	0.81	3204	353.759	0.125266

34	0.80	3260	341.089	0.171682
35	0.89	3264	434.826	0.101434
36	0.78	3372	341.694	0.127154
37	0.84	3276	384.543	0.104185
38	0.86	3164	455.573	0.122421
39	0.78	3264	349.040	0.107433
40	0.82	3352	320.243	0.149608
41	0.83	3276	383.438	0.150059
42	0.83	3276	351.444	0.160074
43	0.83	3232	344.040	0.123244
44	0.80	3160	350.999	0.143838
45	0.86	3320	320.821	0.130252
46	0.85	3260	374.508	0.178213
47	0.77	3244	341.819	0.150872
48	0.82	3364	337.087	0.118769
49	0.83	3276	365.558	0.183883
50	0.84	3156	360.757	0.118057
51	0.80	3280	334.949	0.150174
52	0.79	3280	302.722	0.140607
53	0.72	3264	373.848	0.180201
54	0.79	3160	337.643	0.144113
55	0.75	3320	351.729	0.158199
56	0.82	3180	337.459	0.160025
57	0.77	3364	395.361	0.142457
58	0.88	3184	345.235	0.102098
59	0.90	3240	357.207	0.153893
60	0.84	3284	352.697	0.141188
61	0.80	3252	315.433	0.171035
62	0.86	3176	441.416	0.11585
63	0.82	3264	350.015	0.134782
64	0.92	3188	351.909	0.15609
65	0.91	3184	312.737	0.12338
66	0.77	3180	330.375	0.122258
67	0.76	3252	409.373	0.133486
68	0.83	3376	362.720	0.143687
69	0.85	3276	354.196	0.139012

70	0.74	3320	416.449	0.147385
71	0.88	3316	346.117	0.126971
72	0.88	3376	357.537	0.138393
73	0.82	3176	357.536	0.138393
74	0.96	3252	381.528	0.12812
75	0.79	3376	359.628	0.136609
76	0.90	3276	328.809	0.134311
77	0.83	3248	420.336	0.104047
78	0.89	3232	411.221	0.13391
79	0.76	7296	333.591	0.128627
80	1.09	3228	334.769	0.145808
81	1.13	3372	421.005	0.140043
82	1.12	3188	291.000	0.190239
83	0.91	3204	294.754	0.120155
84	0.86	3176	298.047	0.116462
85	0.82	3284	406.569	0.112743
86	0.85	3204	379.672	0.166068
87	0.92	3204	317.674	0.134003
88	0.99	3256	362.870	0.134477
89	0.84	3160	434.912	0.11458
90	1.24	3184	408.199	0.114954
91	0.87	3280	398.361	0.146309
92	1.01	3276	401.631	0.100348
93	1.03	3232	362.419	0.161605
94	0.88	3388	307.567	0.174465
95	0.91	3280	376.915	0.097494
96	1.17	3276	384.522	0.131871
97	0.83	3180	382.480	0.112558
98	0.84	3244	335.056	0.145477
99	0.92	3176	416.437	0.126578
100	0.90	3200	351.071	0.144581

4.4.2 Second Data Set

Table 33 shows setting for the DE-GMDH, that were used during this experiment of the data set [8]. Table 34 shows the results of this experiment. Figures 11, 12, 13 show graphs with

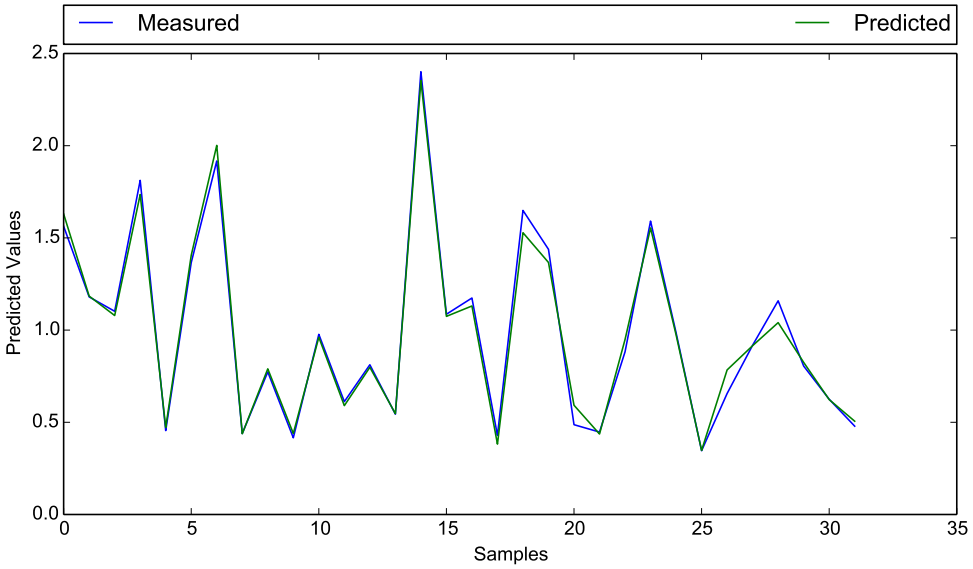

Figure 8: Performance of DE-GMDH on Data Set [13], Result n. 15

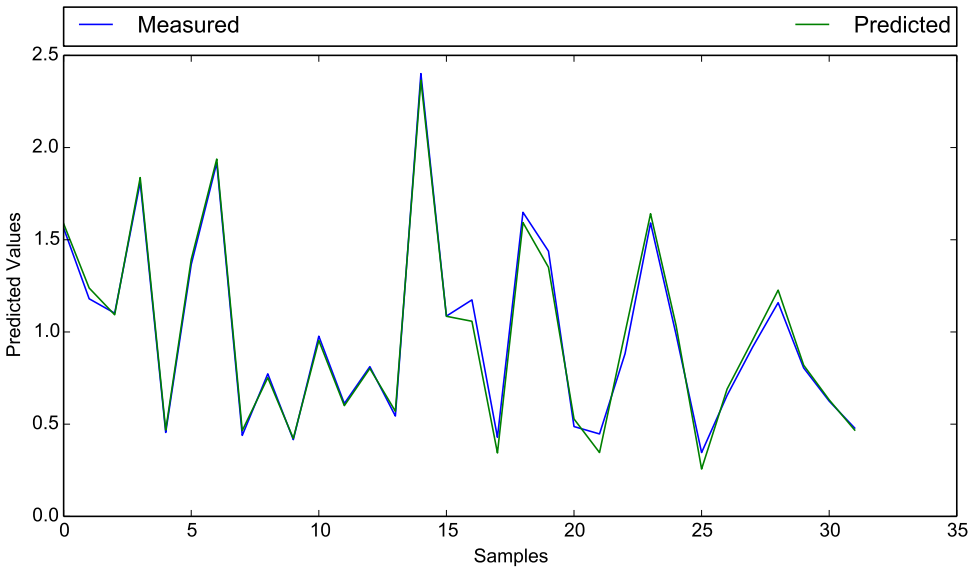

Figure 9: Performance of DE-GMDH on Data Set [13], Result n. 18

measured and predicted values of the runs 30, 58 and 66 respectively.

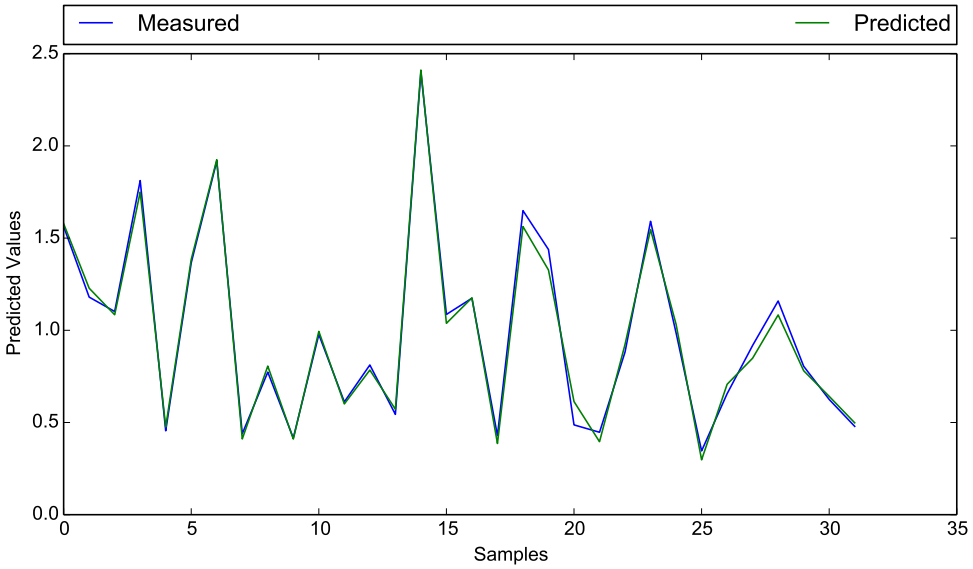

Figure 10: Performance of DE-GMDH on Data Set [13], Result n. 99

Table 33: DE-GMDH Test, Data Set [8], Parameters

Name of the Setting	**Value**
Input Data Columns	7
Training Rows	35
Testing Rows	15
Minimal Network Layers	3
Maximal Network Layers	6
Population Size	300
Number of Generations	200
Number of Threads	2

Table 34: DE-GMDH Test, Data Set[8], Results

Run No.	**Time [s]**	**Memory [kB]**	**Fitness**	**Max Error**
1	3.19	3364	2041.40	0.0793163
2	3.32	3200	1916.24	0.0729995
3	3.41	3280	2002.19	0.0838866
4	4.59	5260	2087.31	0.0736979
5	3.97	3268	2066.39	0.0771928
6	3.56	3260	1806.34	0.0639287
7	3.27	3332	2056.62	0.0794068
8	3.56	3192	1936.25	0.079244
9	3.97	3300	1959.50	0.072832
10	3.57	3256	2021.47	0.0838664
11	3.26	3196	2022.01	0.0783843
12	3.54	3412	1920.36	0.0756203
13	3.85	3300	2101.83	0.072747
14	3.66	3220	1998.90	0.0744233
15	3.62	3224	2093.63	0.06913
16	3.67	3256	1968.87	0.0671543
17	4.55	3388	2123.43	0.0707642
18	3.09	3168	1985.41	0.078678
19	3.49	3200	1953.81	0.0835272
20	3.36	3368	2014.16	0.0799741
21	3.24	3380	2068.18	0.0861551
22	3.01	3276	1922.94	0.082434
23	4.74	3192	2089.57	0.0636792
24	4.62	3288	2176.22	0.0771242
25	3.26	3292	2087.79	0.0736505
26	3.72	3204	2098.13	0.0762446
27	3.14	3280	1883.69	0.0814251
28	3.24	3408	1882.26	0.0811425
29	3.52	3300	1945.50	0.0741548
30	3.46	3260	2021.12	0.0794431
31	2.87	3372	1956.63	0.0799567
32	3.59	3284	2046.95	0.0751383
33	3.08	3204	1948.27	0.0780528

34	3.49	3336	2013.86	0.075197
35	3.53	3292	2081.46	0.081551
36	3.36	3292	2139.23	0.0702226
37	3.88	3272	1940.15	0.0665179
38	4.10	3264	1897.56	0.079643
39	4.02	3276	1967.35	0.0777085
40	3.50	3292	2061.75	0.0722084
41	3.29	3216	2039.04	0.0748774
42	2.97	3256	1984.57	0.0712827
43	3.46	3296	2010.55	0.0853841
44	3.53	3248	2010.72	0.0723205
45	3.41	3196	2024.16	0.0759757
46	3.80	3268	2036.42	0.0705622
47	3.10	3272	1944.32	0.0839333
48	3.42	3296	1977.22	0.0843341
49	3.04	3296	1989.57	0.0874838
50	3.42	3388	2028.72	0.0820741
51	3.13	3284	1907.63	0.0820805
52	3.50	3176	2119.65	0.065644
53	3.27	3368	2044.80	0.0837179
54	3.15	3220	1939.66	0.0846687
55	3.30	3364	2034.25	0.0780035
56	3.33	3412	1921.82	0.0762816
57	3.53	3272	2346.11	0.0820088
58	3.13	3272	1975.44	0.0780723
59	3.28	3292	2149.81	0.0656029
60	3.23	3392	1967.85	0.0755486
61	3.93	3372	2197.16	0.068763
62	3.25	3272	1933.86	0.0658958
63	3.13	3392	1999.71	0.083077
64	4.35	3296	1946.44	0.0767752
65	3.74	3368	2150.45	0.0748482
66	3.59	3280	2065.96	0.0756788
67	3.66	3296	2017.28	0.0771689
68	3.69	3276	2065.49	0.0748053
69	3.42	3396	2048.11	0.0744502

70	3.29	3272	2017.75	0.0736496
71	3.48	3300	2069.53	0.0744044
72	3.13	3248	2129.22	0.0737626
73	4.00	3208	2188.11	0.0604002
74	3.44	3196	2305.96	0.0771828
75	3.35	3260	1900.83	0.0722357
76	3.34	3216	2067.72	0.0661639
77	3.49	5372	2032.07	0.0703624
78	3.47	3372	1939.54	0.0847816
79	3.19	3296	2304.87	0.072162
80	3.59	3296	2080.29	0.0775607
81	3.65	3264	2006.39	0.0809953
82	3.21	3408	2120.61	0.0709433
83	3.32	3196	1842.07	0.0778607
84	3.46	3288	1970.74	0.0723547
85	3.73	3384	2060.38	0.0613087
86	3.43	3368	2096.64	0.0792035
87	3.32	3268	1866.82	0.0804082
88	3.69	3284	2250.79	0.0737537
89	3.49	3380	2139.83	0.0828615
90	3.60	3264	2164.19	0.072987
91	3.19	3176	2043.14	0.0791809
92	3.45	3296	2159.01	0.0691032
93	3.47	3172	2087.77	0.0816217
94	3.64	3280	1992.42	0.0860745
95	3.24	3336	2006.95	0.0795708
96	3.42	3264	1872.99	0.0713104
97	3.26	3292	2026.10	0.0771668
98	3.41	3304	1927.81	0.0832533
99	3.45	3204	1864.08	0.0789134
100	3.51	3260	2027.26	0.068186

4.4.3 Third Data Set

Table 35 shows setting for the DE-GMDH, that were used during this experiment of the data set [11]. Table 36 shows the results of this experiment. Figures 14, 15, 16 show graphs with

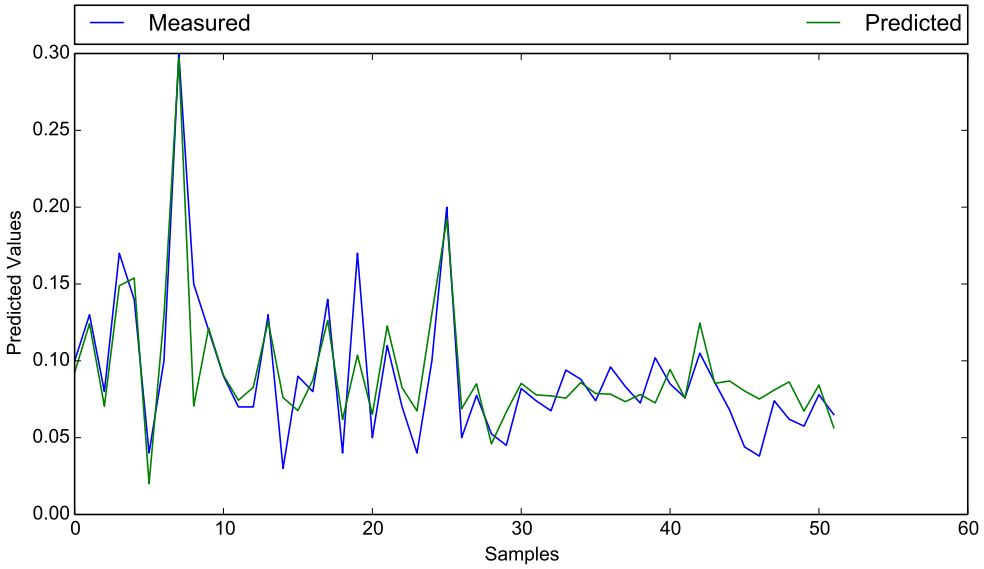

Figure 11: Performance of DE-GMDH on Data Set [8], Result n. 30

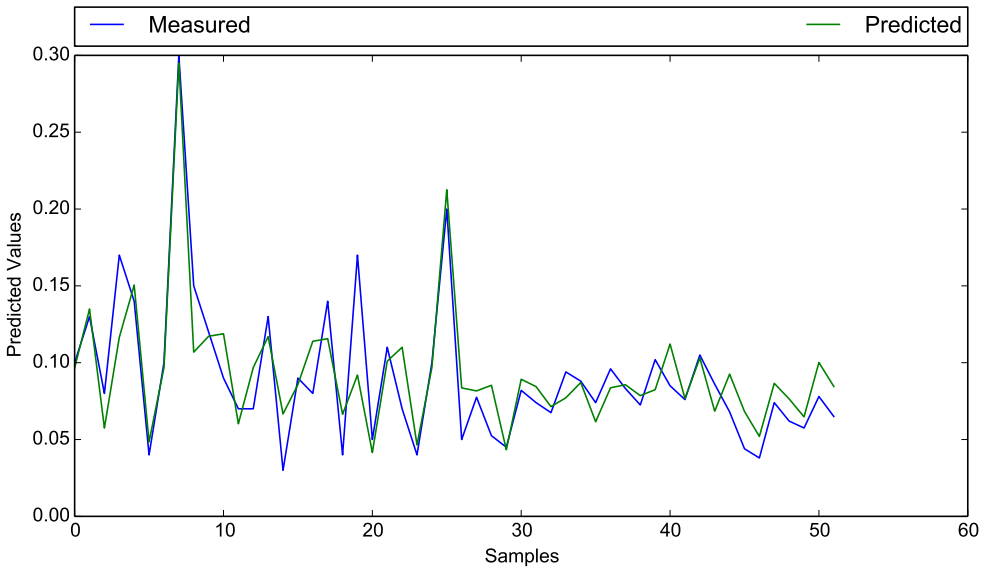

Figure 12: Performance of DE-GMDH on Data Set [8], Result n. 58

measured and predicted values of the runs 27, 48 and 94 respectively.

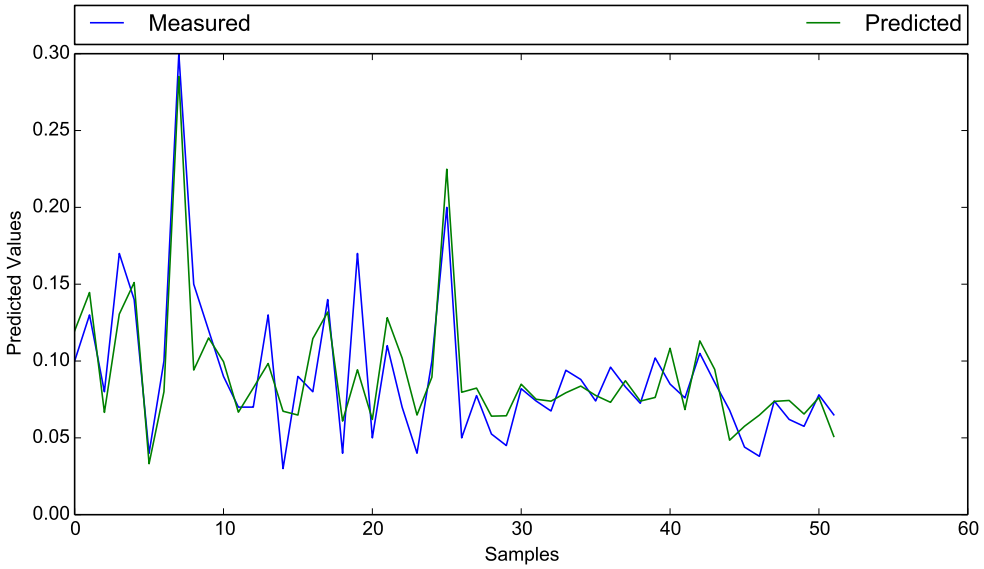

Figure 13: Performance of DE-GMDH on Data Set [8], Result n. 66

Table 35: DE-GMDH Test, Data Set[11], Parameters

Name of the Setting	**Value**
Input Data Columns	8
Training Rows	48
Testing Rows	20
Minimal Network Layers	3
Maximal Network Layers	5
Population Size	200
Number of Generations	300
Number of Threads	2

Table 36: DE-GMDH Test, Data Set[11], Results

Run No.	**Time [s]**	**Memory [kB]**	**Fitness**	**Max Error**
1	11.05	3312	8.02	0.982515
2	9.42	3308	9.51	0.792771
3	12.17	3348	9.84	0.790907
4	9.11	3192	9.89	1.21925
5	9.22	3404	10.42	0.741038
6	11.20	3292	11.31	0.723873
7	8.92	3260	9.45	1.06642
8	9.29	3396	11.62	0.874577
9	12.57	3304	11.16	1.05915
10	8.51	3292	10.18	0.866129
11	8.46	3232	9.15	1.03542
12	9.90	3420	8.72	1.05225
13	8.86	3420	8.62	0.770817
14	8.19	3212	8.89	1.06146
15	8.94	3380	11.10	0.776602
16	10.76	3208	9.39	0.909112
17	9.42	3232	9.23	0.876594
18	8.64	3420	9.13	0.907707
19	11.86	3280	9.73	0.762563
20	8.91	3308	8.50	0.776128
21	8.33	3228	9.69	0.826108
22	8.76	3312	11.13	1.04738
23	8.55	3292	10.25	1.06475
24	8.40	3376	8.11	1.04192
25	8.85	3380	8.81	1.09471
26	8.82	3408	12.47	0.855537
27	8.53	3236	10.23	0.868345
28	8.36	3304	9.75	1.1451
29	8.97	3212	8.70	0.947506
30	8.85	3232	8.72	0.965071
31	9.19	3308	10.09	0.826108
32	8.71	3304	10.91	0.806484
33	9.05	5276	8.33	0.902174

34	8.58	3304	8.81	0.868797
35	9.00	3260	8.92	0.998098
36	8.57	3384	10.09	0.826108
37	8.17	3348	9.95	1.00589
38	8.57	3192	10.94	0.829831
39	8.65	3276	10.51	1.07898
40	8.20	3232	11.24	0.703747
41	8.21	3288	8.50	0.830847
42	8.40	3212	9.78	0.798889
43	8.73	3292	8.89	0.890912
44	8.41	3308	9.75	0.928692
45	8.75	3420	8.25	0.869296
46	8.77	3308	11.34	0.745576
47	8.89	3392	10.67	0.921538
48	8.65	3292	9.84	0.818856
49	8.21	3304	9.08	0.950988
50	8.32	3300	7.95	0.811601
51	8.95	3188	9.55	0.900837
52	8.37	3288	11.57	0.848294
53	8.85	3404	9.32	0.737687
54	8.81	3288	9.88	0.826109
55	8.57	3308	11.25	0.962365
56	8.87	3304	13.34	0.679892
57	8.32	3212	9.74	0.883564
58	8.17	3380	9.90	1.01949
59	9.28	3424	9.08	0.817781
60	9.29	3264	9.47	0.926157
61	8.79	3212	10.31	0.742625
62	8.79	3308	10.94	0.823457
63	8.42	3208	8.92	0.998098
64	8.69	3280	10.03	0.923275
65	8.48	3208	11.47	0.84364
66	8.27	3380	9.06	0.791702
67	8.56	3288	9.90	0.767436
68	8.79	3408	9.19	0.810517
69	8.66	3292	11.53	0.892511

70	8.59	5276	8.98	0.834783
71	8.72	3424	10.82	0.855282
72	8.46	3188	10.32	0.85769
73	8.41	3420	9.01	1.29643
74	7.96	3344	8.83	0.944639
75	9.09	3232	10.64	0.820314
76	8.38	3420	10.27	0.887867
77	8.38	3308	9.90	0.888998
78	8.28	3392	10.69	0.934538
79	8.75	3288	8.11	0.920469
80	8.49	3352	10.12	0.922684
81	8.13	3288	10.54	0.750269
82	8.63	3308	9.07	0.796465
83	8.12	3284	9.07	1.25922
84	8.98	3292	9.25	0.893134
85	9.30	3236	9.98	0.984178
86	10.67	3344	9.22	0.899675
87	8.97	3236	9.77	0.87256
88	8.32	3284	8.83	1.14174
89	8.68	3264	9.87	0.940894
90	9.61	3352	8.20	1.05465
91	13.14	3384	9.31	0.938758
92	8.46	3184	8.86	0.857977
93	8.70	3188	11.14	0.791459
94	8.98	3312	9.00	1.00022
95	8.48	3192	9.87	0.850252
96	9.15	3356	10.06	0.842232
97	8.31	3404	10.31	0.758566
98	8.86	3408	9.95	1.04081
99	7.65	3412	10.70	0.995424
100	8.55	3228	10.78	0.784684

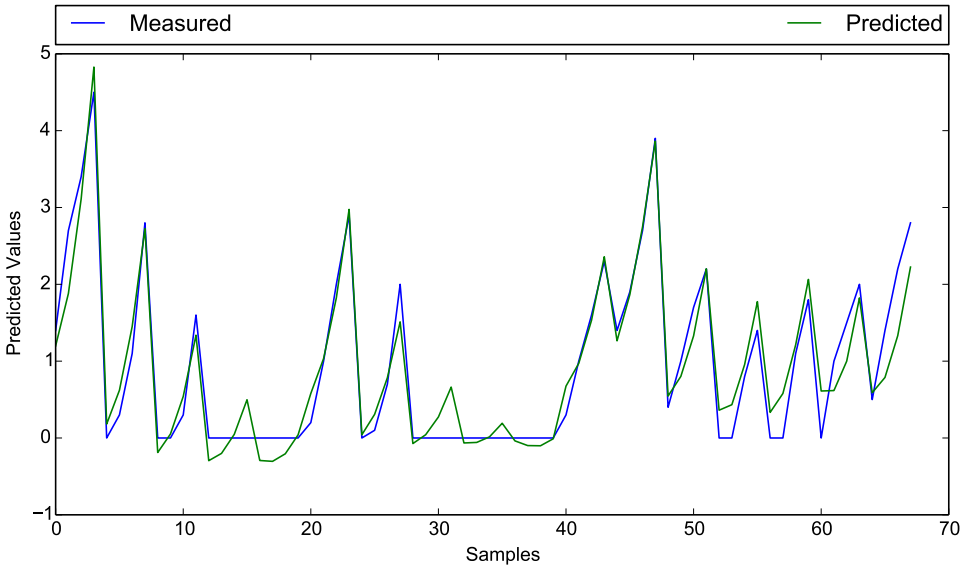

Figure 14: Performance of DE-GMDH on Data Set [11], Result n. 27

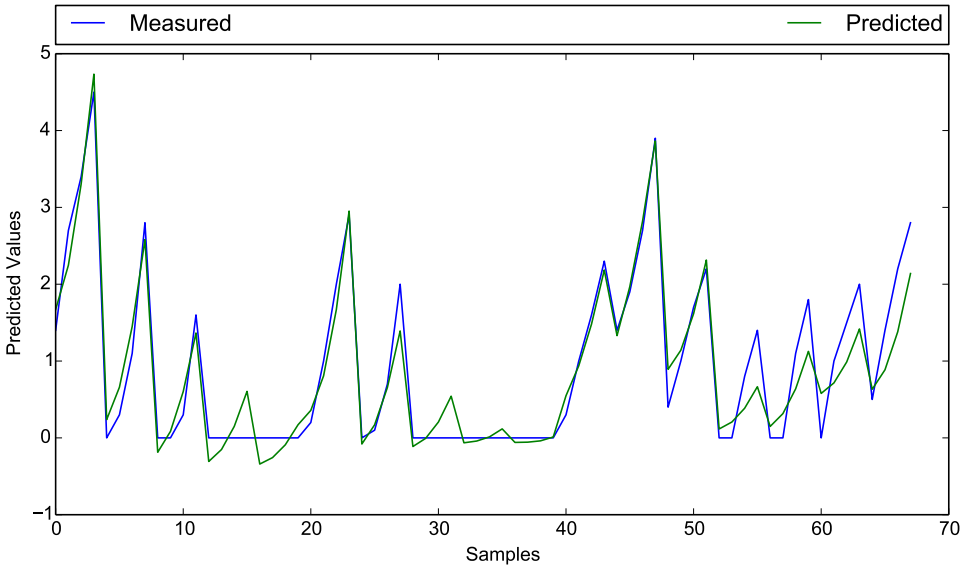

Figure 15: Performance of DE-GMDH on Data Set [11], Result n. 48

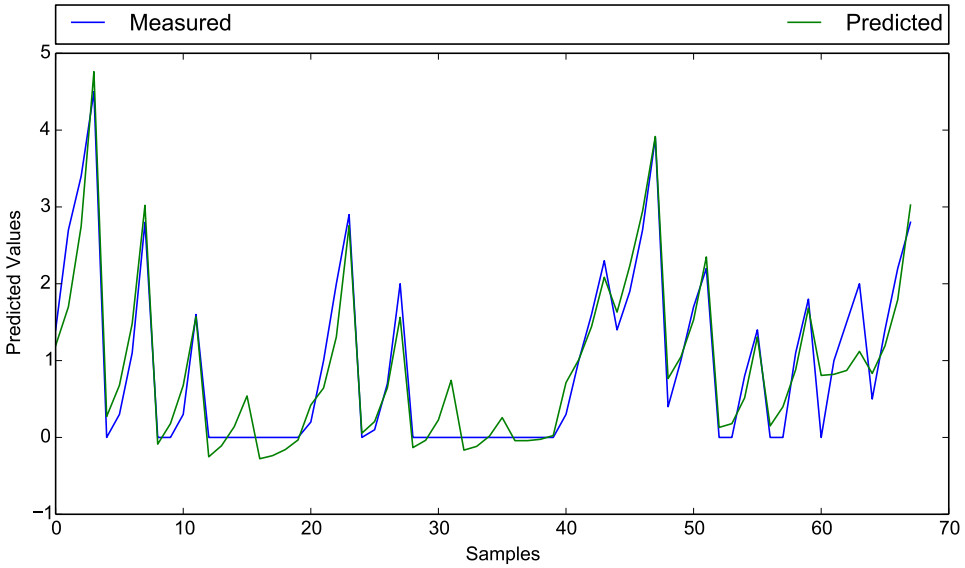

Figure 16: Performance of DE-GMDH on Data Set [11], Result n. 94

5 Analysis

5.1 Network Structure Optimization

5.1.1 Efficiency Experiment

Experiment in Table 2 clearly shows that Network Structure Optimization takes much less time, than the processing of the network.

If we look at the processing time and number of non-redundant nodes relation, we will find out that the correlation is almost linear. This leads to the conclusion that by removing n redundant nodes, the processing should roughly take n nodes processing time less.

Table 2 also shows that networks with a higher number of estimated layers, have a bigger difference between all and non-redundant nodes. The input data had 12 columns and best individual had 6 estimated layers (Table 1), which should give the results in 6 estimated layers some validity, but networks with much higher number of layers probably have this large difference between between all and non-redundant nodes because there is not enough input data combinations in the 1st network layer to create network model with lower redundancy.

For example, networks with 6 estimated layers, that took $0.00436619s$ to optimize, have significant difference between all and non-redundant nodes $15036 - 13576 = 1217$, which is roughly between the numbers of non-redundant nodes in Network Models with 2 and 3 estimated layers (706 and 1645, respectively). These networks took $0.055459s$ and 0.148753 to process, respectively. Both of these times are significantly higher than 0.00557357. That clearly shows, that time was saved by removing redundant nodes, from that network.

5.1.2 Parallelization Experiment

Tables 5 and 6 show that parallelization of the Network Structure Optimization is not suitable for networks with less than 10 estimated layers. But it also shows that time efficiency of the parallelization grows with the network complexity, which makes this parallelization relevant for complex networks.

Tables 7 shows that generally speaking our thread wrapper was able to handle usage of more threads better, but for example for two threads Open MP version had better result until network models with 19 estimated layers, where our thread wrapper was better. It is hard to say which version of the threads performed better.

5.2 Calculation Orders: Recursive vs Layer-by-Layer

5.2.1 Normal Versions

The main reason why Recursive calculation order was considered, was that it should have a better maximal number of results in the memory than Layer-by-Layer. Table 9 clearly shows that this assumption holds. It also shows that time that the calculations took is initially slightly

better for Layer-by-Layer order, but as network complexity grows Recursive order obtains better times. These tributes make Recursive order better than Layer-by-Layer.

5.2.2 Parallelized Versions

Results shown in Tables 9, 10, 11, 12, 13 shows that:

- Maximal number of results in the memory for Layer-by-Layer order is roughly the same.
- Maximal number of results in the memory for Recursive order is slightly increasing with increasing number of threads, but even with this slight increase of maximal number of results, Recursive order still has better memory management than Layer-by-Layer.

Tables 14 and 15 shows that both orders have good time efficiently when parallelized. Layer-by-Layer is superior, when it comes to time efficiency on the relatively small networks. Its time efficiency on the more complex networks is roughly the same as the efficiency of the Recursive order, but because Recursive order has better calculation time, the actual calculation time of Layer-by-Layer order is worse, than that of the Recursive order (Tables 10, 11, 12, 13).

These results make Recursive parallelization superior, because as the next section show, the most time efficient way of small networks parallelization is Parallelized DDE, therefore parallelization of the calculation order is needed only for more complex networks.

Tables 22 and 23 show, that our thread wrapper has better performance, than Open MP for versions of both Layer-by-Layer and Recursive calculation order, that use 2 threads. But it also under performs Open MP when more threads is used, in case of Layer-by-Layer calculation order, performance difference is almost negligible, but in case of the Recursive calculation order, the difference goes up to 3.13%.

Tables 25 and 26 show, that GCC Open MP version has slightly better times with higher number of threads, but results on these tables differ from each other and differences are relatively small, therefore these results should be used as an indication not a proof.

It is also worth mentioning, that only POSIX thread version of our thread wrapper and POSIX thread version of Open MP were compared, both compiled with GCC. Which makes scope of these tests rather narrow. Tests with with different compilers were not carried out, because they are beyond scope of this thesis.

5.3 DDE vs Parallelized DDE vs Parallelized Recursive

5.3.1 First Experiment

Values in Table 28 shows that:

- Memory usage is very similar and mainly low, therefore there is no need to be concerned about it.

- Time results of the Multithreaded DDE with single threaded Recursive calculation order for fitness evaluation were better than the time results of the Single threaded DDE with multithreaded Recursive calculation order for fitness evaluation.

  This results were to be expected, because as we already know that multithreaded Recursive calculation order has slightly bad time efficiency for small network models. And Multithreaded DDE was created because we expected that it will have good time performance for small network models.

These observations show that the Multithreaded DDE is better suited for the populations that contain individuals with relatively small network models.

5.3.2 Second Experiment

Values in Table 30 show that:

- Memory of the Single threaded DDE with multithreaded Recursive calculation order for fitness evaluation is only slightly higher than the memory usage of Single threaded DDE with single threaded Recursive calculation order for fitness evaluation.

  This was to be expected, because it corresponds with what we already know about the Multithreaded Recursive calculation order.

- Memory usage of the Multithreaded DDE with single threaded Recursive calculation order for fitness evaluation is higher than memory usage of the Single threaded DDE with multithreaded Recursive calculation order for fitness evaluation.

  This was also expected, because all the individuals have high memory usage and 4 threads have to process them at the same time.

- Time results of the Multithreaded DDE with single threaded Recursive calculation order for fitness evaluation were worse than the time results of the Single threaded DDE with multithreaded Recursive calculation order for fitness evaluation.

  These results imply that multithreaded Recursive calculation order has good time efficiency for large network models. Also the fact that every single fitness evaluation takes considerable time means, that Multithreaded DDE has a chance of a relatively long wait for the last thread to finish. It is true, that with larger populations this time loss may be less significant.

These observations show that that Multithreaded DDE is not suitable for the populations that contain individuals with relatively high network models. For such populations, single DDE with multithreaded Recursive calculation is preferred.

5.4 Verification of the DE-GMDH implementation

The aim of the thesis is to create an efficient DE-GMDH implementation, and not the piece-wise improvement of the various components of the GMDH model. This section outlines the verification of the developed model.

5.5 First Data Set

Minimal-maximal error in Table 32 is 0.097494 and maximal-maximal error is 0.190239, range of the measured results is $[0.346, 2.401]$, which makes that minimal and maximal error around 4.74% and 9.26% of that range, respectively. Figures 8, 9, 10 also show that measured and predicted values never exceed that maximal-maximal error.

5.6 Second Data Set

Minimal-maximal error in Table 34 is 0.0604002 and maximal-maximal error is 0.0874838, range of the measured results is $[0.03, 0.3]$, which makes that minimal and maximal error around 22.37% and 32.4% of that range, respectively. Figures 11, 12, 13 also show that measured and predicted values never exceed that maximal-maximal error.

5.7 Third Data Set

Minimal-maximal error in table 36 is 0.679892 and maximal maximal error is 1.29643, range of the measured results is $[0, 4.5]$, which makes that minimal and maximal error around 15.1% and 28.8% of that range, respectively. Figures 14, 15, 16 also show that measured and predicted values never exceed that maximal-maximal error.

6 Conclusion

6.1 Development of the GMDH architecture in C/C++ framework using the g++ standard

The following aims of the thesis were fulfilled:

1. Implementation of the GMDH version, where an EA is used for the search for the best Network Model and the whole Network Model is defined by the individual in the population is described in the section 3.1). Discrete Differential Evolution was selected as the EA of choice. Sections 1.2 and 1.3 describe the implementation.

2. Network structure optimization (section 3.2), with new structure that holds the information about the network structure (Network Model) was developed. Experiments in the section 4.1.1 were carried out in order to evaluate this optimization, their analysis (section 5.1.1) clearly shows that the optimized network takes less time to process. Furthermore, network structure optimization removes the possibility of more network with the same structure, but different order of inputs of the nodes, to have different results, due to floating point errors.

3. Parallelized version of the network structure optimization was implemented and tested (sections 3.6.1, 5.1.2, respectively). The experimentation analysis shows (section 5.1.2), that parallelization is suitable only for complex networks, which can not be considered a failure, because that is exactly for which is was implemented.

4. Network structure optimization also allowed the creation of the Recursive calculation order (section 3.5), that should have significantly lowers memory usage than classic Layer-by-Layer calculation order. Experiments were carried out to verify it (section 4.2.1) and the obtained results and analysis (section 5.2.1) clearly proved it.

5. Parallelized versions of both Layer-by-Layer and Recursive calculation orders were created and tested (sections 3.6.2, 3.6.3, 4.2.2), analysis of the tests (section 5.2.2) shows that both orders work well on more complex network models and their memory usage is very similar to their single thread variants. Memory usage of the Recursive calculation order increases only slightly with the increasing number of threads, but is still much lower than the memory usage of the Layer-by-Layers calculation order. Therefore, Recursive order was selected as calculation order of choice.

6. Parallelized version of DDE was implemented and tested (sections 3.6.4, 4.3, respectively), analysis of the test results (section 5.3.1) shows that it works well on relatively simple networks, where memory usage is non-issue and evaluation time of single individual is very short, but has non-ideal performance on more complex networks (section 5.3.2), where

memory usage is more important and time to evaluate single individual is significantly longer.

7. To solve the problem of the parallelized version of DDE, the Hybrid DDE Parallelization was proposed (sections 3.7). Conclusions of sections 5.3.1 and 5.3.2 outline the reasons why Hybrid DDE Parallelization was proposed, which is that it uses the correct parallelization for the correct individual.

8. Usage of Hybrid DDE Parallelization makes Layer-by-Layer order inferior in every way, except for its potential for massive parallelization, which was not studied and is beyond the scope of this thesis.

6.2 Implementation of Single Value Decomposition (SVD) as regression analysis technique for the calculations of the coefficients of the neurons

Implementation of Single Value Decomposition for the calculation of the coefficients of the nodes in the GMDH network, was described in [5]. Section 1.1.3 contains a short description of the steps, that needs to be taken for the calculation of the coefficients. The implementation of SVD coefficients calculator itself was designed to be easily replaceable, if other method for the coefficients calculation is to be used.

6.3 Application and verification of the developed GMDH architecture with manufacturing engineering profiling systems

Verification of our implementation was carried out by the experimentation outlined in section 4.4. The analysis of that experiments shows that our implementation was very successful in predicting values of the first data set. The second and third datasets proved to be the most difficult, however it should be noted that experiments in the section 4.4 were not aimed at obtaining the best possible network model, as they were primarily to show that our DE-GMDH implementation with certain setting can consistently deliver result in a certain range (EA is used for the finding of the best network model and the only stop condition is the number of generation, therefore it is not possible to deliver the same results, unless optimal solution is easy to find).

6.4 Further Development

As stated before, our implementation was created with extendability and modularity in mind, therefore future works ca be focused for example on:

- Implementation of different GAs for finding of the best network structure.
- Implementation of different methods to obtain the coefficients of the nodes.
- Massive parallelization of Layer-by-Layer calculation order.

References

[1] John H Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence.* U Michigan Press, 1975.

[2] A. G. Ivakhnenko. The group method of data handling-a rival of the method of stochastic approximation. *Soviet Automatic Control*, 13(3):43–55, 1968.

[3] HR Madala and AG Ivakhnenko. Inductive learning algorithms for complex systems modeling. boca raton. *Ann Arbor, London, Tokyo: CRC Press Inc*, 1994.

[4] Efrñn Mezura-Montes, Jesús Velázquez-Reyes, and Carlos A Coello Coello. A comparative study of differential evolution variants for global optimization. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 485–492. ACM, 2006.

[5] N Nariman-Zadeh, A Darvizeh, and GR Ahmad-Zadeh. Hybrid genetic design of gmdh-type neural networks using singular value decomposition for modelling and prediction of the explosive cutting process. *Proceedings of the Institution of Mechanical Engineers, Part B: Journal of Engineering Manufacture*, 217(6):779–790, 2003.

[6] GC Onwubolu. Optimization using differential evolution. *Institute of Applied Science Technical Report, TR-2001*, 5, 2001.

[7] Godfrey Onwubolu and Donald Davendra. Scheduling flow shops using differential evolution algorithm. *European Journal of Operational Research*, 171(2):674–692, 2006.

[8] SS Panda, AK Singh, D Chakraborty, and SK Pal. Drill wear monitoring using back propagation neural network; data set= 52x7. *Journal of Materials Processing Technology*, 172(2):283–290, 2006.

[9] VV Praveen, S Thangavelu, et al. Performance analysis of variants of differential evolution on multi-objective optimization problems. *Indian Journal of Science and Technology*, 8(17), 2015.

[10] K Price and R Storn. Differential evolution (de). *The URL of which is: http://www1.icsi.berkeley.edu/~storn/code.html*, 2016.

[11] Jing Shi, Jia-Yeh Wang, and CR Liu. Modelling white layer thickness based on the cutting parameters of hard machining; data set= 68x8. *Proceedings of the Institution of Mechanical Engineers, Part B: Journal of Engineering Manufacture*, 220(2):119–128, 2006.

[12] Rainer Storn and Kenneth Price. Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces. *Journal of global optimization*, 11(4):341–359, 1997.

[13] YF Zhang and JYH Fuh. A neural network approach for early cost estimation of packaging products; data set= 32x22. *Computers & Industrial Engineering*, 34(2):433–450, 1998.

A Appendix on CD

The attached CD contains:

sources This folder contains implementation of DE-GMDH.

rawdata This folder contains raw data from the experiments.

thesis This folder contains electronic version of this thesis.